THE NEW EFFICIENT AND ACCURATE ATTRIBUTE-ORIENTED CLUSTERING ALGORITHMS FOR CATEGORICAL DATA

QIN HONGWU

Thesis submitted in fulfilment of the requirements for the award of the degree of Doctor of Philosophy (Computer Science)

Faculty of Computer Systems and Software Engineering
UNIVERSITI MALAYSIA PAHANG

AUGUST 2012

Thesis submitted in fulfilment of the requirements for the award of the degree of Doctor of Philosophy (Computer Science).

SUPERVISORS’ DECLARATION

We hereby declare that we have checked this thesis and in our opinion, this thesis is adequate in terms of scope and quality for the award of the degree of Doctor of Philosophy in Computer Science.

Signature
Name of Supervisor: Professor Dr. Jasni Mohamad Zain
Position: Dean of Faculty of Computer Systems and Software Engineering
Date:

Signature
Name of Co-supervisor: Dr. Tutut Herawan
Position: Lecturer
Date:

STUDENT'S DECLARATION

I hereby declare that the work in this thesis is my own except for the quotations and summaries which have been duly acknowledged. The thesis has not been accepted for any degree and is not concurrently submitted for the award of any other degree.

Signature:	
Name	: QIN HONGWU
ID Number	: PCC09002
Date	:

Dedicated to my wife and daughter for accompanying me all the time

ACKNOWLEDGEMENTS

My most gratitude to Allah SWT, the Almighty for giving me this great chance to enhance my knowledge and to complete this research. May the peace and blessings be upon Prophet Muhammad SAW.

I am very grateful to University Malaysia Pahang for the research funding and opportunity, I will treasure all the time when I was in the University Malaysia Pahang.

I sincerely thank my respected supervisors, Prof Dr. Jasni Mohamad Zain and Dr. Tutut Herawan, for their helpful advice, effective encouragement and constructive guidance throughout this research.

My sincere thanks to my wife for her support and understanding as well as her endless love. To my lovely princess, thank you for the source of motivations and inspirations. Special thanks to my parents, brother and sister for their support and assistance.

I would also like to thank all postgraduate members, all staff in Faculty of Computer Systems and Software Engineering and the CGS staff for their support, cooperation and contribution all the way.

Finally, again I would like to express my thanks to all who involve in completing this research. Only Allah will reciprocate your goodness.

ABSTRACT

Categorical data clustering has attracted much attention recently due to the fact that much of the data contained in today's databases is categorical in nature. Many algorithms for clustering categorical data have been proposed, in which attribute-oriented hierarchical divisive clustering algorithm Min-Min Roughness (MMR) has the highest efficiency among these algorithms with low clustering accuracy, conversely, genetic clustering algorithm Genetic-Average Normalized Mutual Information (G-ANMI) has the highest clustering accuracy among these algorithms with low clustering efficiency. This work firstly reveals the significance of attributes in categorical data clustering, and then investigates the limitations of algorithms MMR and G-ANMI respectively, and correspondingly proposes a new attribute-oriented hierarchical divisive clustering algorithm termed Mean Gain Ratio (MGR) and an improved genetic clustering algorithm termed Improved G-ANMI (IG-ANMI) for categorical data. MGR includes two steps: selecting clustering attribute and selecting equivalence class on the clustering attribute. Information theory based concepts of mean gain ratio and entropy of clusters are used to implement these two steps, respectively. MGR can be run with or without specifying the number of clusters while few existing clustering algorithms for categorical data can be run without specifying the number of clusters. IG-ANMI algorithm improves G-ANMI by developing a new attribute-oriented initialization method in which part of initial chromosomes is generated by using the attributes partitions. Four real-life data sets obtained from University of California Irvine (UCI) machine learning repository and ten synthetically generated data sets are used to evaluate MGR and IG-ANMI algorithms, and other four algorithms are used to compare with these two algorithms. The experimental results show that MGR overcomes the limitations of MMR and the average clustering accuracy is improved by 19% (from 0.696 to 0.83), at the same time maintains the highest efficiency. IG-ANMI greatly improves the efficiency of G-ANMI (improved by 31% on the Zoo data set, 74% on the Votes data set, 59% on the Breast Cancer data set, and 3428% on the Mushroom data set) as well as the clustering accuracy of G-ANMI (the average clustering accuracy on four UCI data sets is improved by 10.6%, from 0.815 to 0.901), at the same time maintains the highest clustering accuracy. IG-ANMI has obvious advantage against G-ANMI on large data sets in terms of clustering efficiency as well as clustering accuracy. In addition, both of MGR and IG-ANMI have good scalability. The running time of MGR and IG-ANMI algorithms tend to vary linearly with the increase of the number of objects as well as the number of clusters.

ABSTRAK

Pengelompokan data kategori telah menarik banyak perhatian baru-baru ini disebabkan kebanyakan data yang terkandung di dalam pangkalan data hari ini adalah data kategori dalam alam semula jadi. Algoritma untuk mengelompokkan data kategori telah banyak dicadangkan, di mana algoritma pengelompokan hierarki berorientasikan sifat, Min-Min Roughness (MMR) mempunyai kecekapan tertinggi tetapi mempunyai ketepatan pengelompokan yang rendah, sebaliknya, algoritma pengelompokan genetik Genetic-Average Normalized Mutual Information (G-ANMI) mempunyai ketepatan tertinggi dalam pengelompokan, manakala kecekapan kelompok yang rendah. Tesis ini mendedahkan kepentingan ciri-ciri dalam pengelompokan data kategori, dan kemudian menyiasat batasan algoritma MMR dan G-ANMI, seterusnya mencadangkan algoritma baru pengelompokan berorientasikan sifat memecah-belahkan hierarki dipanggil Mean Gain Ratio (MGR) dan algoritma pengelompokan genetik yang lebih baik dipanggil Improved G-ANMI (IG-ANMI) bagi data kategori. MGR mengandungi dua langkah iaitu memilih sifat kelompok dan memilih kelas kesetaraan pada sifat kelompok. Konsep teori maklumat berdasarkan purata nisbah keuntungan dan entropi kelompok digunakan untuk melaksanakan kedua-dua langkah ini. MGR boleh dijalankan dengan atau tanpa menyatakan bilangan kelompok, manakala beberapa algoritma kelompok yang sedia ada bagi data kategori hanya boleh dijalankan tanpa menyatakan bilangan kelompok. Algoritma IG-ANMI meningkatkan prestasi G-ANMI dengan membangunkan satu sifat pengawalan baru yang berorientasikan kaedah di mana sebahagian daripada kromosom awal dijana dengan menggunakan pembahagian sifat. Empat set data sebenar yang diperolehi dari UCI dan sepuluh sintetik set data yang dihasilkan sendiri digunakan untuk menilai algoritma MGR dan algoritma IG-ANMI, dan empat algoritma lain digunakan untuk membandingkan dengan kedua-dua algoritma tersebut. Keputusan eksperimen menunjukkan bahawa MGR mengatasi batasan MMR dan purata ketepatan pengelompokan meningkat sebanyak 19% (0,696-0,83), pada masa yang sama mengekalkan kecekapan yang tertinggi. IG-ANMI memperbaiki kecekapan G-ANMI (bertambah baik sebanyak 31% ke atas set data Zoo, 74% ke atas set data Votes, 59% ke atas data Kanser Payudara, dan 3428% ke atas set data Mushroom) dan juga pada ketepatan kelompokan G-ANMI (purata ketepatan kelompokan pada empat data UCI dipertingkatkan sebanyak 10.6%, daripada 0.815 ke 0.901), pada pada masa yang sama mengekalkan kelompok ketepatan tertinggi. IG-ANMI mempunyai kelebihan yang jelas terhadap G-ANMI pada set data yang besar dari segi kecekapan kelompok serta ketepatan pengelompokan. Di samping itu, kedua-dua MGR dan IG-ANMI mempunyai penskalaan yang baik. Masa larian untuk MGR dan IG-ANMI berubah selari dengan pertambahan bilangan objek dan juga kelompok.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AIB	Agglomerative Information Bottleneck
ANMI	Average Normalized Mutual Information
CACTUS	CAtegorical ClusTering Using Summaries
CE	Conditional Entropy
CNC	Current Number of Clusters
CORE	CORrelated-Force Ensemble
DCF	Distribution Cluster Feature
G-ANMI	Genetic-Average Normalized Mutual Information
GA	Genetic Algorithm
GR	Gain Ratio
IB	Information Bottleneck
IG	Information Gain
IG-ANMI	Improved Genetic-Average Normalized Mutual Information
KDD	Knowledge Discovery in Databases
MDA	Maximum Dependency Attributes
MGR	Mean Gain Ratio
MMR	Min-Min-Roughness
ROCK	RObust Clustering using linKs
SOM	Self-Organizing Map
SSE	Sum of the Squares of Error
STIRR	Sieving Through Iterated Relational Reinforcement
TCSOM	Transactions Clustering using Self-Organizing Map
UCI	University of California Irvine

CHAPTER 1

INTRODUCTION

In this chapter, the background of the research is outlined, followed by problems statement, research objectives, research outcomes, and lastly, the thesis organization.

1.1 BACKGROUND

In this section, three terms related to our research are briefly described, including Knowledge Discovery in Databases (KDD), Data Mining, and Clustering. Figure 1.1 illustrates the relation between them.

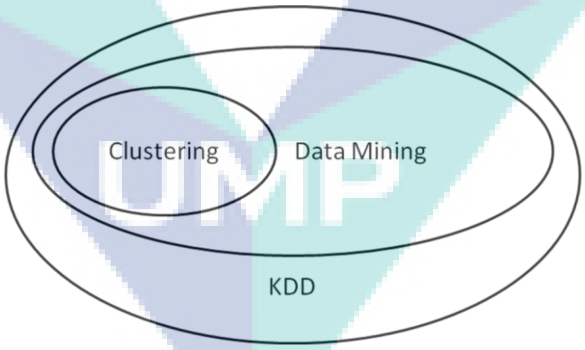


Figure 1.1: The relation between KDD, Data Mining and Clustering

1.1.1 KDD

With the rapid development of information and communication technology, human beings have accumulated tremendous amounts of data. Contributing factors include popular use of internet, ubiquitous data collection and storage devices including digital camera, mobile phone, bar code scanner, and various inexpensive disks with size of gigabytes or terabytes, and the computerization of many fields such as industry, scientific, medicine, business, education, and so on. The explosive growth of data has generated a big challenge to data processing techniques: how to extract useful information and knowledge from the vast amounts of data (Han and Kamber, 2006). This leads to a significant research on the field of information processing, that is, KDD (Piatesky-Shapiro et al., 1996; Sever, 1998; Duntsch and Gediga, 2000; Atkinson-Abutridy et al., 2004). KDD aims to discover some novel, valid, useful, and understandable rules or patterns in large data sets (Piatesky-Shapiro et al., 1996). The process of KDD consists of a series of transformation steps, as is shown in Figure 1.2.

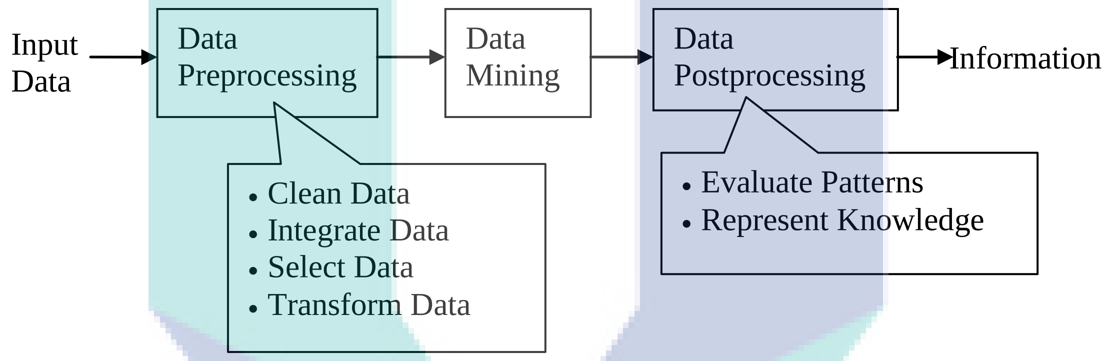


Figure 1.2: The process of KDD

Adapted from: Tan et al. (2006)

First, data preprocessing transforms the original data into an appropriate form according to the need of following processes. The steps involved in data preprocessing include:

- Clean Data: This step removes noise in the raw data.
- Integrate Data: This step fuses data from multiple sources.
- Select Data: This step selects data related to the knowledge discovery task from the database.
- Transform Data: This step transforms or consolidates data into appropriate format for mining by summarizing or aggregating data. Data reduction may

also be performed to gain a subset of the original data without sacrificing its integrity.

After the preprocessing, some data mining methods are employed to extract potential data patterns. The choice of methods usually depends upon the goal of the KDD task. The selection of algorithms generally depends upon the type of the data set and the attribute values.

Generally, a specific user is only interested in a small part of the patterns produced by the data mining methods. Thus, a postprocessing procedure is required. The steps involved in postprocessing include:

- Evaluate patterns: This step indentifies the valid and useful patterns according to some measures given by user.
- Represent Knowledge: The mined knowledge is represented to the user by using some information representation and visualization techniques.

1.1.2 Data Mining

Data mining (Han and Kamber, 2006; Tan et al., 2006; Han et al., 2011; Fu, 2011; Liao et al., 2012) is the process of mining potential rules or patterns in vast amount of data stored in data warehouses, large databases, or other knowledge repositories. There are two categories of tasks in data mining: descriptive mining task and predictive mining task. The former reveals the general character of the data in the database. The latter builds inference model and conducts prediction on the current data (Han and Kamber, 2006). Concretely, four of the core data mining tasks are described below.

i. Classification

The process of data classification consists of two steps. The first step builds a model, which is usually called classifier by applying some classification algorithm on a training data set with the known class labels. Classifier characterizes data classes and distinguishes them. Classification is a typical supervised learning process because the class label of each training data is previously known. There are many methods for constructing classifier, such as decision tree, *k*-nearest neighbor classification, Bayesian classification, artificial neural network, and support vector machines. The second step uses the derived classfier to predict the class of current data whose class label is unknown.

ii. Association analysis

Association analysis is used to discover interesting relationships concealed in large databases. The discovered relationships are typically represented as association rules. Association analysis consists of two steps. The first step finds all the itemsets whose frequence is equal to or greater than the predefined support count threshold. These itemsets are called frequent itemsets. The second step produces strong association rules in terms of the found frequent itemsets which satisfy minimum support threshold as well as minimum confidence threshold. Association analysis can be applied to many fields such as market basket transactions, web mining bioinformatics, medical diagnoses, and so on.

iii. Cluster analysis

Unlike classification which analyzes class-labeled data objects, clustering is a typical unsupervised learning process, that is, the class labels of data objects involved in clustering are previously unknown when the clustering algorithm is being executed; even in most cases the number of clusters is also unknown in advance. The data objects are clustered in terms of the rule of making the intracluster similarity (the similarity between the objects in the identical cluster) maximum and making the intercluster similarity (the similarity between the objects in the different clusters) minimum (He et al., 2005a). In other words, an object is much more similar to the objects in the identical cluster than to the objects in other clusters. Clustering has been used to group sets of related customers, documents, genes, communities in social network and so on.

iv. Outlier detection

Outlier detection is the task of identifying data objects whose characteristics are significantly different from the rest of the data objects. Such data objects are known as outliers. Outliers usually are simply considered as noise data to be discarded in many data mining algorithms. Nevertheless, the unusual events might be more significant than the more usually appearing ones in some specific applications such as deceit detection. The goal of outlier detection is to discover the real outliers and avoid falsely labeling normal objects as anomalies. Outlier detection has been widely applied to the detection of deceit (Ngai et al., 2011), network intrusions, and unusual patterns of disease.

1.1.3 Clustering

As mentioned in Section 1.1.2, clustering is one of the core tasks of data mining. It can reveal the potential patterns and distributions in the objective data by grouping the similar objects together. Clustering plays a very important role in many data processing tasks, for example large data sets segmentation, summarize data, unsupervised classification, and so on (Halkidi et al., 2001).

Clustering techniques have been applied to many domains such as gene data analysis, mobile computing, social networks, medicine, document analysis, security assessment in power systems, management, and image and video anlysis. For example, Hung and Peng (2011) developed algorithm Time Clustering to cluster call detail records for mining mobile user movement patterns. Jiang et al. (2004) analyzed many clustering methods for complicated gene related data. Zhao and Zhang (2011) introduced a new clustering method for extracting community structure from social networks. Wong et al. (2002) proposed a clustering method which can be applied on tissues segmentation in a medical imaging approach. Kalogeratos and Likas (2011) proposed a robust clustering method called k-synthetic prototypes to implement document clustering. Feng et al. (2011) proposed a graph based model to cluster Chinese blogs by using embedded sentiments. Saglam et al. (2006) developed a clustering method to segment customers of a company based on the customer and transaction attributes. Kalyani and Swarup (2011) introduced K-means clustering approach into the evaluation of power system. Given operating condition and contingency, the state of a power system is classified to secure or insecure. Cheng and Leu (2009) proposed a k-prototypes clustering algorithm to handle the classification problems of construction management. Based on cluster analysis, Mathieu and Gibson (2004) proposed a method for large scale research and development planning. Cheng et al. (2011) applied spectral clustering into 3D human posture segmentation with surface normal constraint. Vretos et al. (2011) proposed a face clustering algorithm based on mutual information to analyze movie contents. Huang et al. (2011) proposed a robust clustering algorithm to do image segmentation in L*a*b* color space.

1.2 PROBLEMS STATEMENT

Most of aforementioned clustering algorithms only can be applied on numerical data. A common way to measure the similarity between numerical data points is to define distance functions by employing the inherent geometric attributes in numerical data. Nevertheless, there exists a lot of categorical data in the databases (Parmar et al., 2007) which is usually characterized by a set of descriptive attributes. There is no order between the attribute values on these descriptive attributes. Therefore, distance measure can not be defined between categorical data values. For example, suppose a data set which stores some information about customers is considered. As is shown in Table 1.1, each customer is described by the attributes “age”, “sex”, “income”, and “investments”. Obviousy, it is difficult to define the distance or similarity between the values “youth” and “senior”, or between the customers 3 and 4. Therefore, those clustering algorithms dealing with numerical data can not be directly employed to categorical data clustering. Recently, researchers have paid much attention on categorical data clustering.

Table 1.1: An instance of categorical data set

ID	Age	Sex	Income	Investments
1	youth	male	high	stocks
2	middle-aged	female	high	stocks
3	youth	female	medium	stocks
4	senior	male	medium	bonds
5	middle-aged	male	low	bonds
6	senior	female	medium	bonds

Formally, the problem of categorical data clustering is defined as follows. Given a data set D containing objects O_1, …, O_N, every object O_j (j = 1…N) is described by d categorical attributes, that is, O_j is a d-dimensions vector $O_j = (O_j^1, ..., O_j^d)$. With a specified integer k, the objects are broken into k groups G_1, …, G_k, toward the goal of the minimization/maximization of a criterion (objective function). That is, the problem of categorical data clustering is an optimization problem. Unfortunately, this optimization problem is NP-complete (Barbara et al., 2002). Therefore most researchers resort to heuristic methods to solve it, such as k-modes (Huang, 1998), CAtegorical ClusTering Using Summaries (CACTUS) (Ganti et al., 1999), RObust Clustering using linKs (ROCK) (Guha et al., 2000), COOLCAT (Barbara et al., 2002), Squeezer (He et al., 2002), Transactions Clustering using Self-Organizing Map (TCSOM) (He et al., 2005b), ccdByEnsemble (He et al., 2005a), MMR (Parmar et al., 2007), k-Average Normalized Mutual Information (k-ANMI) (He et al., 2008) and so on. Heuristic methods tend to find local optimal clustering (Deng et al., 2010), thus a few researchers try to solve the problem of categorical data clustering by direct optimization, such as genetic clustering algorithms ALG-RAND (Cristofor and Simovici, 2002) and G-ANMI (Deng et al., 2010), both of them use genetic algorithm (GA) to find globally optimal clustering.

Evaluating these clustering algorithms for categorical data is a non-trivial task (He et al., 2008). At present, the most widely used criteria for evaluation include clustering accuracy (Huang, 1998) and clustering efficiency. Clustering accuracy measures the quality of the results of clustering algorithms. It has been pointed out aht the higher the value of clustering accuracy, the better the results of clustering (He et al., 2008). Clustering efficiency is usally measured by the running time of an algorithm. Obviously, the more running time an algorithm takes on a data set, the lower efficiency it has. Experimental results (Guha et al., 2000; Cristofor and Simovici, 2002; Parmar et al., 2007; Deng et al., 2010) have shown that heuristic algorithms have low clustering accuracy while have high efficiency in comparison with direct optimization based algorithms, conversely, direct optimization based algorithms have high clustering accuracy while have low efficiency in comparison with heuristic algorithms. Hence, new clustering algorithms for categorical data with high efficiency as well as high clustering accuracy are needed.

One of the heuristic algorithms is MMR (Parmar et al., 2007). MMR is based on the rough set theory. It first chooses a partitioning attribute with MMR value, and then split the set of objects into two clusters on the selected partitioning attribute. Repeat the above process on the current longest clusters until reaching the specified number of clusters (The details of MMR algorithm are described in Section 2.3.8). The above clustering procedure can be organized top-down as a tree structure; hence MMR belongs to hierarchical divisive algorithm. In addition, all the partition operations are performed on the attribute; therefore, inherently MMR is an attribute-oriented hierarchical divisive algorithm. MMR is one of the most efficient algorithms among the aforementioned algorithms; however, it has some inherent limitations. First, it is biased toward the attribute with the smallest value domain size or with the most unbalanced partition as determining the partitioning attributes. Second, it selects the longest cluster to split in each iteration. However, the real clusters are not always embedded in the attribute with the smallest value domain size or with the most unbalanced partition, and selecting the longest cluster to split is not always consistent with the natural distribution of clusters. Therefore, these limitations result in low clustering accuracy of MMR algorithm. On the contrary, G-ANMI, one of the direct optimization based algorithms, has the highest clustering accuracy among the existing algorithms for categorical data clustering; however, it has very low efficiency. Inherently, G-ANMI is a genetic clustering algorithm, that is, genetic algorithm is used to find globally optimal clustering in this algorithm. The low efficiency of G-ANMI is mainly caused by genetic algorithm in which lots of iterations are needed to find globally optimal clustering (Deng et al., 2010).

Based on the drawbacks of MMR and G-ANMI algorithms, there is a need for improving these two algorithms. Therefore, two research questions have been proposed:

- How to develop an attribute-oriented hierarchical divisive algorithm to cluster categorical data that attains high clustering accuracy, at the same time maintains the highest efficiency among the existing algorithms for categorical data clustering?
- How to develop a genetic algorithm to cluster categorical data that attains high efficiency, at the same time maintains the highest clustering accuracy among the existing algorithms for categorical data clustering?

In addition, for existing algorithms for categorical data clustering, users have to specify the cluster number before running them. However, it is a difficult task for user to know the exact number of clusters in a data set in advance. Hence, the third research question is:

- How to develop categorical data clustering algorithms which can be run without specifying the number of clusters?

1.3 RESEARCH OBJECTIVES AND SCOPE

This research embarks on the following objectives:

i. To develop an attribute-oriented hierarchical divisive algorithm to cluster categorical data that attains high clustering accuracy, at the same time maintains the highest efficiency among the existing algorithms for categorical data clustering, and can be run without specifying the number of clusters.

ii. To develop a genetic algorithm to cluster categorical data that attains high efficiency, at the same time maintains the highest clustering accuracy among the existing algorithms for categorical data clustering.

iii. To evaluate the proposed algorithms on some real-life data sets and some synthetically generated data sets as well, and to do a comparison between the proposed algorithms with the baseline algorithms in terms of the clustering accuracy and efficiency.

The scope of this research falls within categorical data clustering using attribute-oriented methods.

1.4 RESEARCH OUTCOMES

The following are the research outcomes:

i. The design and development of a novel attribute-oriented hierarchical divisive algorithm for categorical data clustering.

ii. The design and development of an improved genetic algorithm for categorical data clustering.

1.5 THESIS ORGANIZATION

The rest of this thesis is divided into 5 chapters and organised as follows:

Chapter 2:	The basic knowledge about clustering and existing literatures about categorical data clustering are reviewed in this chapter. It covers the components needed in the design of clustering algorithms, some earlier clustering algorithms, entropy based algorithms, k-means like algorithms, and optimization based algorithms.
Chapter 3:	This chapter analyzes the significance of attributes in categorical data clustering, and proposes a novel attribute-oriented hierarchical divisive clustering algorithm for categorical data using information theory, termed MGR. An illustrative example is described to show how MGR algorithm works. Finally, it analyzes the limitations of MMR algorithm, conducts the comparison between MGR and MMR, and presents the advantage of MGR over MMR algorithm.
Chapter 4:	This chapter analyzes the reason for low efficiency of G-ANMI algorithm, and proposes an improved genetic clustering algorithm for categorical data, termed IG-ANMI. A new attribute-oriented initialization method of IG-ANMI algorithm is described and an illustrative example is described to show how it works.
Chapter 5:	This chapter describes the experiments design and experimental results of MGR and IG-ANMI algorithms including the clustering accuracy, clustering efficiency, scalability, and the running results of MGR algorithm without specifying the number of clusters, and compares them with other four algorithms in terms of clustering performance and efficiency.
Chapter 6:	This chapter describes the conclusions and future work.

CHAPTER 2

LITERATURE REVIEW

2.1 INTRODUCTION

This chapter introduces the basic knowledge about clustering and several popular clustering algorithms for categorical data. The chapter is organized as follows:

- Section 2.2 introduces some basic steps to develop clustering process and the components needed in the design of clustering algorithms.
- Section 2.3 describes several popular clustering algorithms for categorical data.

2.2 CLUSTERING

Before describing particular clustering algorithms for categorical data, the basic steps to develop clustering process and the components needed in the design of clustering algorithms including data types, dissimilarity measures, objective functions, membership and the categorization of clustering methods are briefly discussed.

2.2.1 Basic Steps in Clustering Process

Generally, developing a clustering process includes the following basic steps (Piatesky-Shapiro et al., 1996):

- Preprocessing. It is necessary to preprocess the raw data before using them in clustering task, such as feature selection, handling missing value.
- Clustering algorithm selection. Select or design a clustering algorithm

appropriate for the objective data set.

- Evaluation of clustering results. Employing some mostly-used criteria or measurements to evaluate the clustering results, such as clustering error, adjusted rand index, clustering efficiency and so on. These criteria or measurements should be irrespective of specific clustering algorithms (Rezaee, Lelieveldt and Reiber, 1998).
- Interpret clustering results. In most cases, the domain experts are needed to give a meaningful interpretation of the clustering results. This process is usually accomplished with the help of other experimental analysis.

2.2.2 Data Types

The data processed by cluster analysis probably derive from various applications. Each object usually is characterized by *m* attributes, that is, each object is a *m*-dimensional vector. Thus, data type refers to the type of attribute value. In general, there are the below five types of attribute value:

- Numerical
  Numerical data can either be continuous or discrete. There exists a natural order between two numerical values on the same attributes such that distance measures based on geometric attributes can be defined. Length, temperature, pressure, velocity and weight are typical examples of numerical data.
- Binary
  A binary attribute takes on values either 1 or 0. Value 1 indicates the presence of the attribute, while value 0 indicates the absence of the attribute. For example, an animal has value 1 on binary attribute *feather* means that the animal has feather, has value 0 means that the animal has not feather.
- Categorical
  Categorical attribute is usally regarded as the generalization of binary attribute which can take on more than two values. There is no inherent order or similarity between the multiple values of an attribute. Let us take *color* as an example, which is a categorical attribute that might take four values: black, white, green, and blue. Words, integers (Notice that such integers do not

represent any specific ordering), and symbols can be used to denote categorical attribute values.

- Ordinal

  Ordinal data can either be continuous or discrete. Ordinal attributes focus on the relative ordering of the values rather than the actual magnitude. For example, in a sports game the relative ranks (e.g. champion, runner-up) are usually cared much more than the actual score of an athlete.

- Mixed

  In some real-life data sets, objects might be characterized by attributes with different data types. Generally, a data set can include all of the aforementioned four data types. For example, the profile of customers consists of categorical attributes such as nationality and education background as well as numerical attributes like income.

2.2.3 Dissimilarity Measures

Many clustering algorithms require the definition of dissimilarity measure, which examines the closeness between the data objects. Many methods have been proposed to define dissimilarity measures. For different dissimilarity measures, the clustering results may be different. It depends on one’s motivation and data types to choose the appropriate measure. For some algorithms, a dissimilarity matrix is constructed to store the similarities of all pairs of data objects. In this section, some commonly used dissimilarity measures will be discussed.

i. Dissimilarity Metrics for Numerical Data

The dissimilarity between two numerical data objects is usually calculated in terms of the distance between them. Euclidean distance is the most widely-used distance measure whose definition is as follows.

$$dis(S,T) = \sqrt{\sum_{i=1}^{n}(s_i - t_i)^2} \tag{2.1}$$

where $S = (s_1, s_2, \ldots, s_n)$ and $T = (t_1, t_2, \ldots, t_n)$ are two n-dimensions objects.

Manhattan distance is also a widely-used measure, which is defined as

$$dis(S, T) = |s_1 - t_1| + |s_2 - t_2| + \ldots + |s_n - t_n| \tag{2.2}$$

ii. Dissimilarity Metrics for Binary Data

There are two types of binary attributes, symmetric and asymmetric. If two values are equally important for a binary attribute, namely they have the same weight, the binary attribute is symmetric. For example, attribute *gender* is symmetric since values male and female have the same weight. For the objects S and T with symmetric attributes, a simple mismatch coefficient can be used to assess the dissimilarity between them.

$$dis(S,T) = \frac{u+v}{r+u+v+w} \tag{2.3}$$

where r, u, v, and w are defined as follows:

w: the number of attributes on which objects S and T have value 0.

r: the number of attributes on which objects S and T have value 1.

v: the number of attributes on which object S has value 0 and object T have 1.

u: the number of attributes on which object S has value 1 and object T have 0.

If two values have different weight for a binary attribute, the binary attribute is asymmetric. For example, negative and positive results of a medical diagnosis have different influences on the disease verification. For asymmetric attribute, positive matches (two 1s) are usually regarded more important than negative matches (two 0s). Thus, the number of negative matches, w, can be excluded in the calculation. Jaccard coefficient is the most popular measure which examines the proximity between two data objects. Its definition is as follows:

$$J(S,T) = \frac{r}{r+u+v} \tag{2.4}$$

iii. Dissimilarity Metrics for Categorical Data

The dissimilarity between two categorical data *S* and *T* is usually calculated as the proportion of mismatches:

$$dis(S,T) = \frac{q}{n} \tag{2.5}$$

where *n* is the number of attributes and *q* is the number of attributes on which *S* and *T* have the distinct value, that is, the number of mismatches.

Converting the categorical attributes to binary attributes is an alternative method. Concretely, each of the values in an attribute is set as a new binary attribute. Then the dissimilarity measures for categorical data discussed above can be used.

iv. Dissimilarity Metrics for Ordinal Data

The way of calculating the dissimilarity between objects with ordinal attributes is similar to that of calculating the dissimilarity between objects with numerical attributes. First, the range of each attribute is mapped onto [0.0, 1.0] ([*m*, *n*] denotes an interval, including all the values between *m* and *n*) to make each attribute has the same weight, and then dissimilarity is computed using any of the distance measure for numerical data.

v. Dissimilarity Metrics for Mixed Data

For mixed data, a desirable approach is to perform a single clustering, processing all data types together. First, the dissimilarities between the objects *S* and *T* on each type of attribute are computed according to the corresponding dissimilarities measures discussed above, and then map the dissimilarities onto the same interval [0.0, 1.0]. Finally, the average of the normalized dissimilarities is calculated as the dissimilarity between the objects *S* and *T*.

2.2.4 Objective Functions

For many clustering algorithms, clustering is treated as an optimization problem whose aim is to make an objective function minimization or maximization. The definitions of objective function vary with algorithms. For example, some algorithms use the sum of the squares of error (SSE) as objective function (Huang, 1998 and San et al., 2004). Some graph based clustering algorithms define an objective function relavant to the Normalized Cut in a graph (Shi and Malik, 2000 and Flake et al., 2004). Information-theoretic algorithms usually define an objective function based on the entropy of clusters (Barbara et al., 2002; Andritsos, 2004 and Andritsos et al., 2004), conditional entropy of partitions (Cristofor et al., 2002) or maximize the average normalized mutual information (He et al., 2008 and Deng et al., 2010). Some algorithms use the links between data objects to define an objective function (Guha et al., 1999).

2.2.5 Membership

In general, clustering algorithms assume that objects belong to one single cluster only, however, in some cases, different clusters may overlap each other, and hence some objects may have multiple memberships. Many fuzzy theory based algorithms have been proposed to handle the problem (Huang and Ng, 1999; Kim et al., 2004 and Gan et al., 2009). In the framework of fuzzy clustering each object belongs to multiple clusters with different memberships rather than exactly belonging to one cluster. The membership is usually defined as a function of dissimilarities. For each object, memberships among different clusters must sum to 1. For example, a data set of 4 objects is partitioned into 3 clusters using some fuzzy theory based clustering algorithm and a membership matrix $W = \{w_{ij}\}$ is obtained as is shown in Table 2.1, where w_{ij} denotes the membership of the object O_i to cluster C_j.

Table 2.1: An example of membership matrix

w_{ij}	C_1	C_2	C_3
O_1	0.65	0.10	0.25
O_2	0.20	0.65	0.15
O_3	0.70	0.10	0.20
O_4	0.05	0.10	0.85

From Table 2.1, it can be seen that object O_1 has membership degrees 0.65, 0.10 and 0.25 to clusters C_1, C_2, and C_3, respectively. Since object O_1 has the highest degree of membership to cluster C_1, then object O_1 should belong to cluster C_1. Similarly, objects O_2, O_3, and O_4 should belong to cluster C_2, C_1, and C_3, respectively.

2.2.6 Categorization of Clustering Methods

In general, clustering algorithms can be classified as follows (Jain et al., 1999):

- Hierarchical clustering. These types of algorithms hierarchically decompose the given data set. There are two ways to implement hierarchical methods: agglomerative or divisive. The agglomerative approach is a kind of bottom-up approach. Each object is regarded as a cluster at the beginning, and then the clusters that are close to each other are merged consecutively, until the termination condition is satisfied. The hierarchical divisive approach is a kind of top-down approach. All of the objects are considered in one cluster at the beginning, and then iteratively select a cluster and divide it into some smaller clusters, until the termination condition is satisfied.
- Partitional clustering. These types of algorithms try to directly divide the data set into a group of disjoint clusters. The algorithm requires users to specify the number of clusters at first, and then an initial partition is created by using a partitional approach, next, a relocation process based on the objective criterion is performed to improve the partition successively until there is no improvement in the value of the objective criterion.
- Grid-based clustering. Object space, in this type of methods, is quantized into a grid structure which consists of many cells. Then the whole clustering process is conducted on the grid structure.

- Density-based clustering. This type of clustering is implemented in such a way that extending a given cluster continuously if the density (number of objects) in the neighborhood is greater than the predefined threshold. By using density-based clustering method, clusters with arbitrary shape can be discovered.

2.3 ALGORITHMS FOR CATEGORICAL DATA CLUSTERING

Many algorithms have been proposed for categorical data clustering. Ralambondrainy (1995) proposes a method in which category attributes is converted into binary attributes, that is, each of the categories in an attribute is set as a new binary attribute. These binary attributes is regarded as numeric and handled by k-means (MacQueen, 1967) algorithm. Huang (1997 and 1998) proposed k-modes algorithm to extend the k-means algorithm to categorical data. A new cocept of modes is used to replace the means of clusters in k-means. Subsequently, based on k-modes, many algorithms are proposed including fuzzy k-modes (Huang and Ng, 1999), adapted mixture model (Jollois and Nadif, 2002), tabu search technique (Ng and Wong, 2002), improved initial points determination algorithm for k-modes algorithm (Sun et al., 2002), an extension of k-modes algorithm to transactional data (Giannotti et al., 2002), fuzzy centroids (Kim et al., 2004), initialization methods for k-modes and fuzzy k-modes (Cao et al., 2009 and Bai et al., 2011), attribute value weighting in k-modes (He et al., 2011), a dissimilarity measure for k-modes (Cao et al., 2012), and genetic fuzzy k-modes (Gan et al., 2009). ROCK (Guha et al., 2000) is a hierarchical agglomerative clustering algorithm in which the notion of "links" is defined to measure the closeness between clusters. QROCK (Dutta et al, 2005) improves the efficiency of ROCK algorichm. Sieving Through Iterated Relational Reinforcement) (STIRR) (Gibson et al., 2000) introduces the concept of non-linear dynamic system into categorical data clustering. The objects can be clustered if the dynamic system converges. CACTUS (Ganti et al., 1999) constructs summary information from the data set and uses this summarization to discover clusters. Based on information theory, several algorithms are proposed. COOLCAT (Barbara et al., 2002) explores the connection between clustering and entropy. LIMBO (Andritsos et al., 2004) applies the Information Bottleneck (IB) method to the problem of clustering categorical data. "Best K" (Chen and Liu, 2005) proposes a BkPlot method for determining the best K number of clusters in a categorical data set. The method is implemented with a hierarchical clustering algorithm HierEntro. Recently, several works try to cluster categorical data by using cluster ensemble technique. He et al. (2005a) investigate the commonalities between cluster ensemble and categorical data clustering, and employ some cluster ensemble algorithms to cluster categorical data. Subsequently, the same authors propose k-ANMI (He et al., 2008), which optimizes Average Normalized Mutual Information (ANMI) in a k-means framework. Gionis et al. (2005) cluster categorical data by employing a cluster ensemble algorithm which is based on disagreement measure. MMR (Parmar et al., 2007) applies rough set theory to categorical data clustering. To improve the accuracy and efficiency of clustering, Herawan et al. (2010) propose a rough set based method named Maximum Dependency Attributes (MDA) for selecting clustering attribute. A few researchers try to cluster categorical data by direct optimization. ALG-RAND (Cristofor and Simovici, 2002) and G-ANMI (Deng et al., 2010) employ genetic algorithm to cluster categorical data. TCSOM (He et al, 2005b) algorithm extends traditional Self-Organizing Map (SOM) to cluster binary data. The same authors also propose Squeezer (He et al., 2002) algorithm. Squeezer can be used to cluster categorical data streams since it sequentially deal with each input data object and group it in an appropriate cluster. Chen and Chuang (2004) investigate the correlation between attribute values and propose algorithm CORrelated-Force Ensemble (CORE) based on correlated-force ensemble. There also exist some algorithms focusing on transaction data clustering. Wang et al. (1999) propose the notion of large item and utilize it to cluster transactions. Similarly, another notion named the small-large ratio is presented and used to cluster market basket data (Yun et al., 2001). Yun et al. (2002) apply the item taxonomy to clustering process. Xu and Sung explore the purchase features of customers and propose a caucus based algorithm. Cao and Liang (2011) propose a data labeling method to allocate unlabeled objects into proper clusters.

The following sections will describe the details of some popular algorithms among the above algorithms.

2.3.1 *k*-means, *k*-modes and Fuzzy *k*-modes Algorithms

i. *k*-means algorithm

k-means (MacQueen, 1967) is a typical partitional clustering algorithm. It has been widely used since proposed. The algorithm starts with the selection of initial cluster center, *k* objects is randomly selected as the initial cluster center or mean. Then each of of the rest objects is grouped into the cluster which is the closest to it, in terms of the distance between the cluster center and the object. Next, the new center of each cluster is recomputed. Repeat this procedure until the within-clusters SSE converges. The clustering process can be formulated as follows (Bobrowski and Bezdek, 1991 and Huang 1998):

$$\text{Minimize} \quad P(W,Q)=\sum_{j=1}^{k}\sum_{i=1}^{n} w_{i,j}\, d(X_i,Q_j) \tag{2.6}$$

$$\text{subject to} \quad \sum_{j=1}^{k} w_{i,j}=1,\ 1\le i\le n$$
$$w_{i,j}\in\{0,1\},\ 1\le i\le n,\ 1\le j\le k \tag{2.7}$$

where k is the specified number of clusters, n is the number of objects, X_i is a data object, $Q=\{Q_1, Q_2, \ldots, Q_k\}$ is a matrix of cluster centers, W is an $n\times k$ partition matrix, and $d(.,.)$ is the square Euclidean distance between two data objects.

The optimization problem can be solved by iteratively solving the following two minimization problems:

- Problem P_1: Set $Q=\hat{Q}$ and solve the smaller problem $P(W,\hat{Q})$.
- Problem P_2: Set $W=\hat{W}$ and solve the smaller problem $P(\hat{W},Q)$.

Problem P_1 is solved by

$$w_{i,j}=1 \quad \text{if } d(X_i, Q_j)\le d(X_i, Q_t), \text{ for } 1\le t\le k$$
$$w_{i,t}=0 \quad \text{for } t\ne j \tag{2.8}$$

Let $Q_j = (q_{j,1}, \ldots, q_{j,m})$, problem P_2 is solved by

$$q_{j,s} = \frac{\sum_{i=1}^{n} w_{i,j} x_{i,s}}{\sum_{i=1}^{n} w_{i,j}} \tag{2.9}$$

for $1 \leq s \leq m$, and $1 \leq j \leq k$.

ii. *k*-modes

The *k*-means algorithm can only be used to cluster numerical data since the similarity between data objects and cluster centers is measured by Euclidean distance. To extend the *k*-means algorithm to categorical data, Huang (1998) proposed the *k*-modes algorithm in which Euclidean distance is replaced with a simple matching dissimilarity measure, and the means of clusters are replaced with modes.

- **Dissimilarity measure**

Given two objects *X*, *Y*, suppose they are characterized by *m* categorical attributes, *k*-modes algorithm counts the total mismatches of the corresponding attribute values of the two objects as the dissimilarity measure between *X* and *Y*, which is denoted as $d_c(X,Y)$ and defined as follows,

$$d_c(X,Y) = \sum_{l=1}^{m} \delta(x_l, y_l) \tag{2.10}$$

where

$$\delta(x_l, y_l) = \begin{cases} 0 & (x_l = y_l) \\ 1 & (x_l \neq y_l) \end{cases} \tag{2.11}$$

- **Mode of a Set**

Given a set of categorical objects $X = \{X_1, X_2, \ldots, X_n\}$, each of which is characterized by *m* categorical attributes. The vector $Q = [q_1, q_2, \ldots, q_{\mathrm{m}}]$ that minimize

$$D(X,Q)=\sum_{i=1}^{n} d_c(X_i,Q) \tag{2.12}$$

is called the mode of X. It has been proved that each element in the mode takes the most frequent value of the corresponding attribute in the set.

- **Algorithm**

Replacing the Euclidean distance in Equation (2.6) with the dissimilarity measure defined in Equation (2.10), the new objective function is obtained.

$$P(W,Q)=\sum_{j=1}^{k}\sum_{i=1}^{n}\sum_{l=1}^{m} w_{i,j}\,\delta(x_{i,l},q_{j,l}) \tag{2.13}$$

Accordingly, problem P_1 is solved by using the new dissimilarity measure, problem P_2 is solved by using modes of clusters. Figure 2.1 lists the steps of *k*-modes algorithm in detail.

1. Select *k* initial modes, one for each cluster.
2. Allocate an object to the cluster which mode is the nearest to it according to Eq. (2.10). Update the mode of the cluster after each allocation.
3. After all objects have been allocated to clusters, retest the dissimilarity of objects against the current modes. If an object is found such that its nearest mode belongs to another cluster rather than its current one, reallocate the object to that cluster and update the modes of both clusters.
4. Repeat 3 until no object has changed clusters after a full cycle test of the whole data set.

Figure 2.1: The *k*-modes algorithm

iii. Fuzzy *k*-modes

Following the fuzzy *k*-means-type process, *k*-modes algorithm can be extended to fuzzy *k*-modes. The objective of the fuzzy *k*-modes clustering is to find *W* and *Q* that minimize

$$P(W,Q)=\sum_{j=1}^{k}\sum_{i=1}^{n} w_{j,i}^{\alpha} d_c(X_i,Q_j) \tag{2.14}$$

subject to Eq.(2.7), where $\alpha > 1$ is the weighting exponent, $d_c(.,.)$ is defined in Eq.(2.10), *W* = (w_{ji}) is the $k \times n$ fuzzy membership matrix, and *Q* ={Q_1, Q_2, …, Q_k} is the set of cluster centers. Note that $\alpha = 1$ gives the hard *k*-modes clustering, i.e., the *k*-modes algorithm.

Let *Q* ={Q_1, Q_2, …, Q_k} be fixed, then the fuzzy membership matrix *W* is updated by

$$w_{j,i}=\begin{cases} 1 & if \quad X_i = Q_j; \\ 0 & if \quad X_i = Q_h, h \neq j; \\ \dfrac{1}{\sum_{h=1}^{k}\left[\dfrac{d_c(X_i,Q_j)}{d_c(X_i,Q_h)}\right]^{\frac{1}{\alpha-1}}} & otherwise \end{cases} \tag{2.15}$$

for $1 \le i \le n$, $1 \le j \le k$.

Given the estimate of *W*, the cluster centers is updated as

$$Q_{j,l} = a_{lr} \in DOM(A_l) \tag{2.16}$$

for $1 \le j \le k$, $1 \le l \le m$, where *m* is the number of attributes, *DOM*(A_l) is the domain of attribute A_l, , $r = \arg\max_{1 \le t \le n_l} \sum_{i, x_{il} = a_{lt}} w_{ji}^{\alpha}$.

2.3.2 ROCK and QROCK

i. ROCK

ROCK (Guha et al., 2000) is a hierarchical agglomerative clustering algorithm for categorical data. Traditional agglomerative clustering algorithms only examine the similarity between data objects when objects are considered to be merged into a single cluster. Experiments have shown that traditional methods tend to make errors. ROCK proposes the notions of neighbours and links for categorical data and considers the neighbourhoods of two objects when they are considered to be merged. Two objects can be merged if they have similar neighbourhoods.

- **Neighbours and links**

ROCK first defines the concept of neighbours. Given a similarity function $s(O_i, O_j)$ that measures how close objects O_i and O_j are, O_i and O_j are called neighbours each other if the below inequation holds

$$s(O_i, O_j) \geq \lambda \tag{2.17}$$

where λ is a predefined threshold.

The number of common neighbours between objects O_i and O_j is named the number of links between O_i and O_j, and denoted by $link(O_i, O_j)$. The larger the number of links between two objects, the higher possibility the two objects belong to the same cluster.

- **Criterion function and goodness measure**

ROCK aims to maximize the intra-cluster summation of links, and meanwhile minimize the inter-cluster summation of links. The criterion function is defined as follows:

$$E = \sum_{i=1}^{k} n_i * \sum_{p_q, p_r \in C_i} \frac{link(p_q, p_r)}{n_i^{1+2f(\theta)}} \tag{2.18}$$

where C_i refers to i^{th} cluster, n_i is the size of i^{th} cluster, $n_i^{1+2f(\theta)}$ denotes the expected number of links between objects pairs in C_i.

In order to find a good clustering that maximizes the objective function, ROCK algorithm also define a measure, named goodness measure, to determine the best cluster pair to merge at each step, which is defined as follows.

$$g(C_i, C_j) = \frac{link[C_i, C_j]}{(n_i + n_j)^{1+2f(\theta)} - n_i^{1+2f(\theta)} - n_j^{1+2f(\theta)}} \tag{2.19}$$

where *link*[C_i, C_j] denotes the number of cross links between clusters C_i and C_j, $n_i = |C_i|$ and $n_j = |C_j|$. The value $(n_i + n_j)^{1+2f(\theta)} - n_i^{1+2f(\theta)} - n_j^{1+2f(\theta)}$ denotes the expected number of cross links between objects pairs in clusters C_i and C_j.

- **Algorithm**

The algorithm starts with calculating the number of links between objects. At the beginning, each object forms a single cluster. The algorithm builds a local heap $L[i]$ for each cluster i and a global heap G. Each local heap $L[i]$ contains every cluster which has non-zero links with cluster i. The clusters j in $L[i]$ are ordered in descending order of g(i, j). G contains all the clusters which are ordered in descending order of the max element in each cluster. At each step, select first cluster j in G and the first cluster in $L[j]$ as the best clusters pair to be merged. The related elements in local heap and global heap need to be updated once two clusters are merged. The process iterates until only desired number of clusters left in G. Figure 2.2 describes the ROCK algorithm in detail.

```
Procedure cluster $(D, k)$
begin
    $link$ = compute_links($D$)
    for each $d \in D$ do
        $L[d]$ = build_local_heap($link$, $d$)
    $G$ = build_global_heap($D$, $L$)
    while size($G$) > $k$ do {
        $u$ = extract_max($G$)
        $v$ = max($L[u]$)
        delete($G$, $v$)
        $w$ = merge($u$, $v$)
        for each $x \in L[u] \cup L[v]$ do {
            $link[x, w] = link[x, u] + link[x, v]$
            delete($L[x]$, $u$); delete($L[x]$, $v$)
            insert($L[x]$, $w$, $g(x, w)$); insert($L[w]$, $x$, $g(x, w)$)
            update($G$, $x$, $L[x]$)
        }
        insert($G$, $w$, $L[w]$)
        deallocate($L[u]$); deallocate($L[v]$)
    }
end
```

Figure 2.2: The ROCK algorithm

ii. QROCK

QROCK (Dutta et al, 2005) improves the efficiency of ROCK algorichm. The authors define the notion of link graph and further explore the relationship between the link graph and clustering. Based on the relationship, a quick version of ROCK is proposed.

- **Link graph**

Given a set *S* of objects, the link graph *G* for *S* is builded in such a way that *S* is regarded as the set of vertices, and there is an edge between two vertices if they have non-zero links.

- **The relationship between link graph and the final clusters**

Based on the concept of link graph, the authors prove that the clusters produced by ROCK algorithm are equivalent to the connected components in the link graph *G* if inter-cluster links equal to zero.

- **The algorithm**

The QROCK algorithm starts with the computing of *nblist*[*i*] for each object *i*. *nblist*[*i*] stores the neighbours of data object *i*. Next, repeat the following process on every *nblist*[*i*]: the algorithm take an object *w* from *nblist*[*i*] and merge the connected component containing *w* with the connected components of all other objects in *nblist*[*i*]. Figure 2.3 shows the QROCK algorithm.

```
Input: A set $S$ of objects
begin
    compute $nblist[i]$ for each $i \in S$ using $\lambda$
    for each $i \in S$, initialization($i$)
    for each $i \in S$
    {
        Take a fixed object $w$ in $nblist[i]$
        for each other object $r$ in $nblist[i]$
        {
            $U$ = find_component($w$)
            $V$ = find_component($r$)
            if $U \neq V$
            then merge($U, V$)
        }
    }
end
```

Figure 2.3: The QROCK algorithm

In the above algorithm, the procedure initialization(*i*) creates a connected component that contains only the object *i*, find_component(*w*) returns the connected component which contains object *w*, and merge(*U*, *V*) calculates the union of connected components *U* and *V*.

Compared with ROCK, QROCK algorithm need not to compute the links, build local and global heaps, and sort the heaps, which leads to QROCK are more efficient than ROCK. However, QROCK can not replace ROCK completely because of the different termination condition. ROCK terminates either when *k* clusters are generated or when inter-cluster links equal to zero, however, QROCK terminates only when inter-cluster links equal to zero. Both algorithms produce the same results when ROCK terminates since inter-cluster links equal to zero, but different results when ROCK terminates since the desired number of clusters is reached. Although the authors argue that specifying the desired number of clusters is less practical than specifying the similarity threshold, it can not be guaranteed that the results of QROCK are more meaningful than that of ROCK.

2.3.3 STIRR

Gibson et al. (2000) introduce non-linear dynamical systems into categorical data clustering and propose algorithm STIRR (Sieving Through Iterated Relational Reinforcement). Each attribute category is represented as a weighted vertex in a graph. The set of weights of all the nodes is called a *configuration*. The initial configuration can be either chose uniformly (e.g. all weights set to 1) or, randomly (e.g. each weight set to an independently chosen random value in [0, 1]). The algorithm iteratively updates weight of any single node to change the configuration. The weight of node *v* is updated by applying a combiner function chosen in advance separately to the members of all tuples that contain *v*, and adding the results. Authors' potential choices for combiner function are product operator and addition operator. The dynamical system finally iterates to a stable state called as *basin*.

The algorithm maintains multiple copies of the configuration, w_1, …, w_m for clustering the set of objects. The configuration w_1 iterates to a basin, called principal basin. The basins correspond to other configurations are called non-principal basins. The weights in non-principal basins will divide the attribute values of each attribute into two groups when the fixed point is reached. One group has positive weights and the other has negative weights. Intuitively, these groups correspond to projections of clusters on the attribute. However, a non-trivial post-processing step is required to automatically identify the final clusters, which was not solved in STIRR.

2.3.4 CACTUS

CACTUS (Ganti et al., 1999) constructs summary information from categorcial data set and uses the summarization to discover clusters. By generalizing the definition of a cluster for numerical data, the authors introduce a new definition of a cluster for categorical data. Several basic concepts are defined as follows. Given attribute value a_i in attribute A_i and attribute value a_j in attribute A_j, the number of tuples that have these two values is called *Support* of values pair (a_i, a_j). If their support is greater than a threshold, attribute value pair (a_i, a_j) are said to be *strongly connected*. A *cluster* consists of a set of attribute values each of which is strongly connected to all other attribute values in the cluster. The attribute value pair containing two values from the same attribute is measured by *Similarity*. Given attribute values a_1 and a_2 in attribute A_i, *Similarity* between a_1 and a_2 is defined as the number of such attribute values in other attributes that are strongly connected with a_1 as well as a_2.

CACTUS defines intra-attribute and inter-attribute summaries based on the notions of support and similarity. The similarities between attribute values of the same attribute constitute the *intra-attribute summaries*. All strongly connected attribute value pairs each of which has attribute values from different attributes constitute the *inter-attribute summaries*.

The CACTUS algorithm first calculates all intra-attributes and inter-attributes summaries, which is called summarization phase. Then candidate clusters are generated by extending 2-clusters to 3-clusters and so on, which is called clustering phase. The candidate clusters generated in clustering phase need to be validated. In the validation phase, the support of each candidate cluster is calculated. Only clusters whose support passes the threshold requirement are retained.

2.3.5 Entropy Based Algorithms

COOLCAT and LIMBO are typical entropy based clustering algorithms for categorical data. They are introduced in this section.

i. COOLCAT

COOLCAT (Barbara et al., 2002) first explores the relationship between information entropy and clustering: the entropy of clusters consisting of similar objects is lower than that of clusters consisting of dissimilar ones. The algorithm clusters objects towards the goal of making the expected entropy of the clusters minimization.

- **Problem formulation**

Let D be a data set containing N objects p_1, p_2, …, p_N, where each point is described by M categorical attributes x_1, x_2, …, x_M, and given an integer k, the points would be separated into k groups G_1, G_2,…,G_k , or clusters which have the minimum expected entropy. The algorithm defines the expected entropy of the clusters as follows.

$$\overline{E}(G) = \sum_{i=1}^{k} \left(\frac{|G_i|}{|D|} E(G_i)\right) \tag{2.20}$$

where $G = \{G_1, G_2, \ldots, G_k\}$ represents the clustering. $G_i \subset D$, with the property that $G_i \cap G_j = \varnothing$, for all i, j =1, …, k, $i \neq j$. $E(G_i)$ represents the entropy of the ith cluster. To simplify the calculation, independence of the attributes of the record is assumed, thus, the entropy of the ith cluster is calculated as follows.

$$E(G_i) = E_{G_i}(x_1) + E_{G_i}(x_2) + \ldots + E_{G_i}(x_M) \tag{2.21}$$

where, $E_{G_i}(x_j)$, j=1,…, M, is the entropy of the attribute x_j about the cluster G_i. Unfortunately, the problem is NP-Complete, therefore a heuristic based algorithm is proposed.

- **Algorithm**

The algorithm starts with an initialization step which discovers a set of clusters in a sample set Q taken from original data set D ($|Q| << N$). The initialization step is implemented by finding the *k* most "dissimilar" objects from the sample set *Q*. Concretely, the algorithm first finds two points p_{s_1} , p_{s_2} that have maximum expected entropy $E(p_{s_1}, p_{s_2})$ and places them in two clusters G_1, G_2, respectively. From there, proceeding incrementally, an unprocessed object p_{s_j} that maximizes $min_{i=1\ldots j-1}(E(p_{s_i}, p_{s_j}))$ is chose for the *j*-th cluster.

Next, an incremental step is performed to place the rest of the objects in sample data set *Q* and the remaining objects in data set (*D* - *Q*), in the clusters produced in initialization step. For each of the remaining objects, the algorithm tries to place it in each of the clusters and calculate the corresponding expected entropy, then selects the cluster corresponding to the minimum expected entropy as the final cluster of the objects. Figure 2.4 lists the details of the procedure.

```
1. Given an initial set of clusters $G = \{G_1, G_2, \ldots, G_k\}$
2. Bring objects to memory from disk and for each object $p$ do
3. For $i = 1, \ldots, k$
4.     Place $p$ in $G_i$ and compute $\overline{E}(G^i)$
       where $G^i$ denotes the clustering obtained by placing $p$
       in cluster $G_i$
5.     Let $j = argmin_i(\overline{E}(G^i))$
6. Place $p$ in $G_j$
7. Until all objects have been placed in some cluster
```

Figure 2.4: Incremental step of COOLCAT algorithm

ii. LIMBO

LIMBO (Andritsos et al., 2004) algorithm is an extension of the Agglomerative Information Bottleneck (AIB) (Solnim and Tishby, 1999) algorithm.

- **AIB algorithm**

Let D be a data set containing n objects with m attributes, $G_k = \{g_1, g_2, \ldots, g_k\}$ be a disjoint k-partition (k-clustering) of the objects in D.

AIB is a hierarchical agglomerative clustering algorithm. At the beginning of the algorithm, each object $d \in D$ forms a single cluster. Then the algorithm interates n-k steps to reduce the number of the clusters. Suppose the algorithm is at step n-s+1 with s-partition G_s, two clusters g_i, g_j are selected and merged into a single cluster to generate a new partition G_{s-1}. The selection of clusters g_i and g_j at each step follows the principle that the information loss in transfering from partition G_l to partition G_{l-1} is minimized. The information loss can also be viewed as the increase in the uncertainty. Thus, the objective of AIB algorithm is to minimize the entropy of the clustering, which is the same as the objective function of COOLCAT.

- **Distribution Cluster Feature**

High computational complexity of the AIB algorithm makes it unsuitable for dealing with large data sets (Andritsos et al., 2004). LIMBO proposes a model named Distribution Cluster Feature (DCF) which summarizes information about the clusters for dealing with large data sets. Given a cluster g, the conditional probability of the attribute value, $p(V \mid g)$, and the probability of cluster g, $p(g)$, its DCF is defined as follows:

$$DCF(g) = (p(g), p(V \mid g)) \tag{2.22}$$

DCF is organized as a B-tree in which the leaves denote a clustering of the data set.

- **Algorithm**

The LIMBO algorithm starts with calculating a DCF for each object, and then constructs the DCF tree to summarize the data by inserting these DCFs into it. The next step of the algorithm is to the cluster the objects on DCF tree. The DCFs of the tree leaves are merged by employing the AIB algorithm to generate a specified number of clusters. The authors of LIMBO claim that other clustering algorithm can also be employed in the step. Finally, a scan is performed over the objects and each object is grouped to the cluster whose center is most similar to this object.

2.3.6 k-ANMI

k-ANMI (He et al., 2008) clusters categorical data in a *k*-means framework. It uses average normalized mutual information as the objective function. In order to efficiently implement the algorithm, the concept of histogram of attribute is defined and multiple hash tables (each hash table denotes a histogram) are employed.

- **Objective function**

The mutual information based criterion used in *k*-ANMI borrows from cluster ensemble methods (Strehl and Ghosh, 2002). Given *r* groupings with the *q*th grouping $\lambda^{(q)}$ having *k*(*q*) clusters, the [0,1]-normalized mutual information criteria between grouping $\lambda^{(a)}$ and $\lambda^{(b)}$ is computed as follows:

$$\phi^{(NMI)}(\lambda^{(a)},\lambda^{(b)}) = \frac{2}{n}\sum_{h=1}^{k^{(a)}}\sum_{g=1}^{k^{(b)}} n_g^{(h)} \log_{k^{(a)}*k^{(b)}}\left(\frac{n_g^{(h)} n}{n^{(h)} n_g}\right) \tag{2.23}$$

where n_g is the length of cluster C_g in grouping $\lambda^{(b)}$, $n^{(h)}$ is the length of cluster C_h in grouping $\lambda^{(a)}$, $n_g^{(h)}$ is the number of objects in cluster C_h according to $\lambda^{(a)}$ as well as in cluster C_g according to $\lambda^{(b)}$.

Given a labeling $\overline{\lambda}$, and a set of *r* labelings, Λ, the ANMI between $\overline{\lambda}$ and Λ is calculated as follows:

$$\phi^{(ANMI)}(\Lambda,\overline{\lambda}) = \frac{1}{r}\sum_{q=1}^{r}\phi^{(NMI)}(\overline{\lambda},\lambda^{(q)}) \tag{2.24}$$

Given the desired number of clusters, *k*, the optimal *k*-clustering $\lambda^{(k-opt)}$ should be the clustering that has the highest ANMI with *r* labelings, Λ, that is,

$$\lambda^{(k-opt)} = \arg\max_{\bar{\lambda}} \sum_{q=1}^{r} \phi^{(NMI)}(\bar{\lambda}, \lambda^{(q)}) \tag{2.25}$$

where $\bar{\lambda}$ goes through all possible k-partitions.

- **Data structure**

The main data structure used in *k*-ANMI algorithm is the histograms of attributes. Suppose D is a set of n objects characterized by r categorical attributes A_1, …, A_r. V_i denotes the value set of attribute A_i. q_a denotes the frequency of attribute value $a \in V_i$, that is, the number of objects $O \in D$ with $O.A_i = a$. If attribute A_i has p_i distinct attribute values, the histogram of A_i is defined as the set of pairs: $h_i = \{(a_1, q_1), (a_2, q_2), \ldots, (a_{pi}, q_{pi})\}$. The histogram of dataset D is defined as $H = \{h_1, h_2, \ldots, h_r\}$. *k*-ANMI uses $(k+1)r$ histograms as basic data structure totally, where r histograms are constructed for r attributes, rk histograms for label vector $\bar{\lambda}$.

- **Algorithm**

The *k*-ANMI algorithm starts with an initialization procedure. Each object is put into the closest cluster according to the dissimilarity measure which is calculated as the average distance between this object and the objects in cluster. Next, an iteration phase is performed; each object t is moved to an existing cluster to maximize ANMI until there is no improvement in ANMI for one iteration. Figure 2.5 shows the *k*-ANMI algorithm.

```
Algorithm k-ANMI
Input:   D       // the categorical database
         k       // the number of desired clusters
Output:  clusterings of D

/* Phase 1-Initialization */
Begin
    foreach object t in D
        counter++
        update histograms for each attribute
        if counter<=k then
            put t into cluster $C_i$ where i = counter
        else
            put t into cluster $C_i$ to which t has the smallest distance
        write <t, i>
/* Phase 2-Iteration */
    Repeat
        not_moved =true
        while not end of the database do
            read next object < t, $C_i$ >
            moving t to an existing cluster $C_j$ to maximize ANMI
            if $C_i$ != $C_j$ then
                write <t, j>
                not_moved =false
    Until not_moved
End
```

Figure 2.5: The *k*-ANMI algorithm

2.3.7 Genetic Clustering Algorithms

In this section, two genetic algorithm based methods for categorical data clustering are described, namely ALG-RAND (Cristofor and Simovici, 2002) and G-ANMI (Deng et al., 2010). These two algorithms use genetic algorithm to search the optimal partition of the objects.

i. ALG-RAND

ALG-RAND tries to find a median partition over the space of all partitions of the data objects of the database, which is most similar to all the partitions defined by the attributes of the data set. The algorithm generalizes the concept of classical conditional entropy to evaluate the dissimilarity between two partitions. Since finding the median partition is an NP-complete problem, ALG-RAND uses a genetic algorithm to find an approximative solution.

- **Generalized conditional entropy**

Let f be a generator, R be the set of rows of a table T, and let $\pi = \{B_1,...,B_n\}$, $\sigma = \{C_1,...,C_m\}$ be two partitions of R. The f-conditional entropy of π relative to σ is defined as

$$H^f(\pi \mid \sigma) = \frac{1}{|R|}\sum_{j=1}^{m}|C_j|\sum_{i=1}^{n} f\left(\frac{|B_i \cap C_j|}{|C_j|}\right) \qquad (2.26)$$

Three functions can be used as generator in ALG-RAND, that is, $f_{\text{gini}}(p) = p\text{-}p2$ (the Gini index), $f_{\text{ent}}(p) = \text{-}p\log p$ (the Shannon entropy), and f_{peak} (given by $f_{\text{peak}}(p) = p$ for $0 \le p \le 0.5$ and $f_{\text{peak}}(p) = 1\text{-}p$ for $0.5 \le p \le 1$).

- **Dissimilarity measure**

Let PART(R) denotes the set of partitions of a set R. If f is a generator, $\sigma, \pi \in$ PART(R), then the mapping d^f: PART(R)$\times$ PART(R) $\rightarrow \mathbb{R}$ given by

$$d^f(\pi,\sigma) = H^f(\pi \mid \sigma) + H^f(\sigma \mid \pi) \qquad (2.27)$$

is a definite dissimilarity on PART(R). When π is close to σ, meaning that their classes have many elements in common, then both $H^f(\pi \mid \sigma)$ and $H^f(\sigma \mid \pi)$ are close to 0, so $d^f(\pi,\sigma)$ is close to 0.

- **Genetic algorithm**

Let T be a table with rows (objects) set R and attributes set H, k be the desired number of clusters. The objective of ALG-RAND algorithm is to search for the median partition, that is a partition π such that $|\pi| \leq k$, and π minimizes the sum $\sum_{A\in H} d^f(\pi_A, \pi)$.

A k-chromosome on a table T is function K: $R \rightarrow \{1, \ldots, k\}$. An element of the set $\{1, \ldots, k\}$ is called a class identifier. The partition π_K of the set of rows R determined by the k-chromosome K is $\pi_K = \{C_1, \ldots, C_k\}$, where $C_j = \{r \in R | K(r) = j\}$ for $1 \leq j \leq k$. The chromosomial population consists of k-chromosomes, $K_1, K_2, \ldots, K_M$, where each k-chromosome K_i can be regarded as a sequence of length $N=|R|$ representing a possible assignment of the rows of the table T to the k classes of the partition π_{K_i}. Initially, the chromosomes $K_1, K_2, \ldots, K_M$ are generated using random values between 1 and k.

The idea of the genetic evolution is to modify the chromosomes in the current population by using mutation and crossover as genetic operators such that in the new population chromosomes will be increasingly closer to the median partition, that is, they will summarize better and better the columns of the table T. Figure 2.6 shows the the algorithm in detail.

```
initialize the population of genetic algorithm
while (true)
          compute the fitness of chromosomes in the population;
          if (there has been no relative improvement in best fitness value for
                $N_{max}$ iterations)
          then
                    output the partition of $K_{best}$;
                    exit;
           copy fittest $(1 - r - m)M$ chromosomes to new population;
           select probabilistically max$\{2, rM\}$ chromosomes to cross over;
           apply crossover operator to the selected chromosomes
                  and copy the offspring to the new population;
           select with uniform probability max$\{1, mM\}$ chromosomes to mutate;
           apply mutation operator to the selected chromosomes
                  and copy the modified chromosomes to the new population;
           Use the new population  to replace the old one;
```

Figure 2.6: The pseudo code of the ALG-RAND algorithm

ii. G-ANMI

From the perspective of objective function and searching method, G-ANMI can be considered as a combination of ALG-RAND and k-ANMI algorithms. It uses ANMI, the same one in k-ANMI, as the objective function, and a basic genetic algorithm which works in the same way as the one in ALG-RAND to search globally optimal partition measured by ANMI.

Initially, the algorithm generates a group of partitions of objects and encodes them as chromosomes. These chromosomes form the initial population. Then, ANMI described in Eq. (2.24) is used to calculate the fitness of each chromosome. Based on the fitness values, some genetic operations are employed to generate a new population by changing the chromosomes in the current population. The latest chromosomes are expected to be more similar to the optimal partition than the previous chromosomes. Repeat the above steps until the best fitness has kept invariable in some successive iterations.

2.3.8 MMR

MMR (Parmar et al., 2007) is a rough set based attribute-oriented hierachical divisive clustering algorithm.

- **Definitions**

Given objects set *U*, attributes set *A*, suppose *V* is the set of all attribute values, *Q* is a subset of *A* and *f*: *U*×*A*→*V* is an information function. *Q* defines an indiscernibility relation *IND(Q)* on *U* as

$$IND(Q) = \{(u,v) | ((u,v) \in U \times U) \wedge (\forall q \in Q, f(u,q) = f(v,q))\} \qquad (2.28)$$

Let *U/Q* denote the partition of *U* induced by *IND*(*Q*) and $[u]_Q$ denote the equivalence class containing $u \in U$ in partition *U/Q*. Given a set of objects $T \subseteq U$, the lower approximation and upper approximation of *T* with respect to *Q* are defined respectively as

$$\underline{T}_Q = \{u \mid (u \in U) \wedge ([u]_Q \subseteq T)\} \tag{2.29}$$

$$\overline{T}_Q = \{u \mid (u \in U) \wedge ([u]_Q \cap T \neq \phi)\} \tag{2.30}$$

Given attributes $a_i, a_j \in A$ ($a_i \neq a_j$), let $T(a_i = \gamma)$ denote the subset of objects which have attribute value γ on attribute a_i , $\underline{T}_{a_j}(a_i = \gamma)$ and $\overline{T}_{a_j}(a_i = \gamma)$ denote the lower approximation and upper approximation of $T(a_i = \gamma)$ with regard to a_j , respectively, the roughness of $T(a_i = \gamma)$ with regard to a_j is defined as

$$R_{a_j}(T|a_i = \gamma) = 1 - \frac{\left|\underline{T}_{a_j}(a_i = \gamma)\right|}{\left|\overline{T}_{a_j}(a_i = \gamma)\right|} \tag{2.31}$$

Given attribute $a_i, a_j \in A$, let $V_{a_i} = \{\gamma_1, \gamma_2, ..., \gamma_k\}$, where $k = |V_{a_i}|$, the mean roughness on attributes a_i with respect to a_j is defined as

$$Rough_{a_j}(a_i) = \frac{\sum_{h=1}^{k} R_{a_j}(T|a_i = \gamma_h)}{k} \tag{2.32}$$

Given m attributes, the minimum of the mean roughness on attributes $a_i (a_i \in A)$ with respect to a_j ($1 \leq j \leq m$, $j \neq i$), is called min-roughness of attribute a_i , and defined as,

$$MR(a_i) = Min(Rough_{a_1}(a_i), ..., Rough_{a_j}(a_i), ..., Rough_{a_m}(a_i)) \tag{2.33}$$

Given m attributes, the minimum of min-moughness of these m attributes is called the Min-Min-Roughness (MMR), and defined as,

$$MMR = Min(MR(a_1), ..., MR(a_i), ..., MR(a_m)) \tag{2.34}$$

- **Algorithm**

MMR first chooses a partitioning attribute with MMR value, and then split the set of objects into two clusters on the selected partitioning attribute. Iteratively repeat the process on the current longest clusters until reaching the specified number of clusters. Figure 2.7 describes the details of MMR algorithm.

```
Algorithm: MMR
Input: Data set $U$, and required number of clusters $k$
Output: $k$ clusters on $U$
Begin
    Step 1: Set Longest_Cluster = $U$ (In the following step, clustering will be
            conducted on the Longest_Cluster),
            Set Current Number of Clusters (CNC) = 1.
    Step 2: For each attribute $a_i$, calculate Min-Roughness MR$(a_i)$.
    Step 3: Calculate the Min-Min-Roughness MMR.
    Step 4: Select attribute with the MMR as partitioning attribute.
    Step 5: From the partition defined by partitioning attribute, select the
            equivalence class with the minimum roughness as the splitting
            equivalence class.
    Step 6: Split Longest_Cluster into two clusters: the objects correspond to the
            splitting equivalence class, and the rest objects in Longest_Cluster.
            Store them into Clusters_Table.
    Step 7: Find the current longest cluster in Clusters_Table and assign it to
            Longest_Cluster. Delete the longest cluster from Clusters_Table.
    Step 8: CNC = CNC+1, if CNC < $k$, then go to Step 2, else output the
            clusters in Clusters_Table.
End.
```

Figure 2.7: The MMR algorithm

2.3.9 MDA

MDA (Herawan et al., 2010) is a rough set based method for selecting clustering attribute in categorical data set. It employs the dependency between attributes of the data set to select clustering attribute instead of the roughness between attributes.

- **Definitions**

In information system $S = (U, A, V, f)$, given any two attributes $a_i, a_j \in A$, the dependency of a_i with respect to a_j is denoted by $k_{a_j}(a_i)$ and defined as follows

$$k_{a_j}(a_i) = \frac{\sum_{X \in U/a_i} \left| \underline{a_j(X)} \right|}{|U|} \tag{2.35}$$

Next, given m attributes, max-dependency of attribute $a_i (a_i \in A)$ is defined as

$$MD(a_i) = Max(k_{a_1}(a_i), \ldots, k_{a_j}(a_i), \ldots, k_{a_m}(a_i)) \tag{2.36}$$

where $a_i \neq a_j$, $1 \leq i, j \leq m$.

After obtaining the m values of $MD(a_i)$, $i = 1, 2, \ldots, m$. MDA method selects the attribute with the maximum value of max-dependency as clustering attribute, i.e.

$$MDA = Max(MD(a_1), \ldots, MD(a_i), \ldots, MD(a_m)) \tag{2.37}$$

Based on the dependency of attributes in the rough set theory in information systems, MDA algorithm is given as follows.

- **Algorithm**

```
Algorithm: MDA
Input:    Data set $U$
Output: Clustering attribute
Begin
        Step 1:  For each attribute $a_i \in A$, perform Steps 2 - 4.
        Step 2:  Determine the equivalence classes of $a_i$.
        Step 3:  Calculate $k_{a_j}(a_i)$, $a_j \neq a_i$, $1 \leq j \leq m$.
        Step 4:  Calculate $MD(a_i)$.
        Step 5:  Calculate $MDA$.
        Step 6:  Select attribute with $MDA$ as clustering attribute.
End.
```

Figure 2.8: The MDA algorithm

2.3.10 Squeezer

Squeezer (He et al., 2002) sequentially reads each object from a data set and places it in an appropriate cluster. Since each object is visited once during the whole process, Squeezer can be used to cluster categorical data streams. The algorithm defines a similarity measure between an object and a cluster to determine the final cluster for an object being handed. The similarity measure between a given object *u* and a cluster *G* is computed as:

$$S(u,G) = \sum_{i=1}^{m} \frac{\sup(a_i)}{|G|} \tag{2.38}$$

where a_i is the value of ith attribute of object *u*, *m* is the number of attributes, sup(a_i) denotes the number of value a_i appears in the *i*th attribute of cluster *G*.

For the first object, it forms a single cluster. For the consequent each object, the algorithm first calculate the similarities between the object and the existing clusters, and then selects out the largest value of similarity. If the largest value exceeds the predefined threshold *s*, the object is placed into the cluster corresponding to the largest similarity; else this object forms a new cluster alone. The algorithm halts when all objects in the data set are clustered.

2.3.11 Discussion

k-means like algorithms including *k*-modes and various variants of *k*-modes, have the identical advantage with the *k*-means algorithm, namely they have high efficiency as they handle large data sets. However, they also have the identical disadvantage with the *k*-means algorithm, that is, the clustering results might vary with different initial values of modes (Parmar et al., 2007).

It has been shown that many algorithms such as COOLCAT, LIMBO, and MMR outperform algorithm ROCK. The limitations of ROCK have been pointed out in these works. Firstly, the number of clusters generated by ROCK might exceed the number specified by user. Secondly, ROCK might create a large cluster including objects from many classes. Thirdly, the clustering results produced by ROCK significantly vary with the predefined threshold.

For algorithm STIRR, the data set can be clustered only as the mapped dynamic system converges. Moreover, a non-trivial post-processing step is required to automatically identify sets of closely related attribute values and final clusters, which was not solved in STIRR. Finally, there are many choices of combiner functions; however, STIRR can not guarantee the performance of the system for any combiner function. Rigorous experiments are needed to generate a meaningful clustering.

CACTUS introduces a new definition of a cluster for categorical attributes. A cluster is considered as a high-density node set in which each node pair is strongly connected. The advantage of CACTUS is that it can conduct a subspace clustering since it discovers clusters in subsets of the attributes. Experimental results on synthetic data sets show that CACTUS has better efficiency and scalability than STIRR. However, such a cluster definition requires every node pair to be strongly connected in a cluster. Consequentely, a large number of clusters are likely to be generated and the cluster number may not be close to the desired one in the user perspective (Chen and Chuang, 2004).

Information entropy based algorithms COOLCAT and LIMBO have the same objective function. COOLCAT uses sampling to construct the initial clusters; hence, the clustering results may be affected by the size of sample and the distribution of the real clusters. In addition, the order of processing objects also influences the clustering results so that the authors have to introduce a re-processing phase. The comparative analysis conducted by authors of LIMBO shows that the clustering performance and parameter stability of LIMBO are better than that of STIRR and ROCK.

ALG-RAND, *k*-ANMI, and G-ANMI define the problem of categorical data clustering as optimization problem. On the one hand, the objective functions of these three algorithms are all based on information theory, where ALG-RAND uses conditional entropy while *k*-ANMI and G-ANMI use normalized mutual information. On the other hand, *k*-ANMI works in a *k*-means framework, while ALG-RAND and G-ANMI uses genetic algorithm to find the optimal partition. *k*-ANMI tends to discover local optimal partition, ALG-RAND and G-ANMI try to discover globally optimal partition instead. The authors of *k*-ANMI and G-ANMI have conducted comparative analysis with ALG-RAND and showed that *k*-ANMI and G-ANMI outperform it on clustering accuracy. The authors of *k*-ANMI assert that their algorithm beats other algorithms including *k*-modes, ccdByEnsemble, and Squeezer. While G-ANMI improved clustering accuracy in comparison with heuristic algorithms, low efficiency caused by genetic algorithm is still a large obstacle before it can be widely used.

MMR algorithm has high efficiency due to the fact that it split the objects from the viewpoint of attributes. However, it has some limitations, which will be analyzed in Chapter 3. MDA algorithm improves the accuracy and efficiency of selecting clustering attribute by using the dependency between attributes. However, it can only be used to select clustering attribute rather than to cluster the data set.

Squeezer is suitable for clustering data streams since it scans each tuple only once. However, each dataset need a different threshold makes the selection of threshold a difficult work for users. Although the authors proposed a sampling technique to generate the threshold, it is not necessarily suitable for various datasets (e.g. some datasets are unbalanced). In addition, the algorithm usually produces more clusters in comparison to other algorithms, which probably results in meaningless clusters. For example, the algorithm generates 28 clusters on Congressional Votes dataset; nevertheless, there exists only two real clusters in the dataset.

CHAPTER 3

MGR: A NEW ATTRIBUTE-ORIENTED HIERARCHICAL DIVISIVE CLUSTERING ALGORITHM FOR CATEGORICAL DATA USING INFORMATION THEORY

3.1 INTRODUCTION

In this chapter, a novel hierarchical divisive clustering algorithm for categorical data, named MGR is proposed, which implements clustering from the viewpoint of attributes. The significance of attributes in categorical data clustering is analyzed first, followed by the description of MGR technique, the pseudocode of MGR algorithm, an illustrative example, and comparison with MMR algorithm. The chapter is structured as follows:

- Section 3.2 analyzes the significance of attributes in categorical data clustering.
- Section 3.3 describes the details of MGR technique.
- Section 3.4 describes the MGR algorithm and gives an illustrative example.
- Section 3.5 compares MGR algorithm with MMR algorithm.

3.2 THE SPACE OF ATTRIBUTES PARTITIONS

Most existing clustering algorithms for categorical data focus on the relation between the objects or the relation between an object and clusters during the process of clustering. These relations are usually measured by similarity or dissimilarity. It can be said that these methods are object-oriented. In fact, a data set consists of two elements: objects and attributes. Therefore, besides objects, the attribute is also an important aspect deserving to be considered for clustering. In a categorical data set, each attribute defines a partition of the set of objects, each partition consists of some equivalence classes (an equivalence class is a set of objects which has the same value on an attribute). All the partitions form the space of attributes partitions.

Let us first look at an example. Table 3.1 shows a categorical data set with ten objects and five attributes. The column of real classes implies that the set of objects can be partitioned into three classes. Suppose that the objects in each class are the same while completely distinct from the objects in other classes. A_i, B_i, C_i (i=1,...,5) denote different categories on the *i*th attribute. The requirement now is to cluster the data set without knowing the real classes in advance. Using the object-oriented methods like *k*-modes, *k*-ANMI, the number of clusters has to be specified first and then the processes of initialization, iteration are conducted. Specifying the number of clusters in advance is difficult. Suppose the number of clusters is set to two in this example, the accuracy of clustering would be affected. In fact, from the viewpoint of attributes, it can be seen that each attribute partition the data set in the same way. If such relation between the attributes can be found, a perfect clustering of the data set including three clusters will be obtained by using the partition defined by any attribute without specifying the number of clusters in advance. Therefore, using attribute to cluster the data set in this example is a better way than using the object-oriented methods. The example reveals the potential of the space of attributes partitions for categorical data clustering.

Table 3.1: Example data set with ten objects and five attributes

Objects	**Attribute 1**	**Attribute 2**	**Attribute 3**	**Attribute 4**	**Attribute 5**	**Real classes**
O_1	A_1	A_2	A_3	A_4	A_5	1
O_2	B_1	B_2	B_3	B_4	B_5	2
O_3	B_1	B_2	B_3	B_4	B_5	2
O_4	C_1	C_2	C_3	C_4	C_5	3
O_5	A_1	A_2	A_3	A_4	A_5	1
O_6	B_1	B_2	B_3	B_4	B_5	2
O_7	A_1	A_2	A_3	A_4	A_5	1
O_8	C_1	C_2	C_3	C_4	C_5	3
O_9	B_1	B_2	B_3	B_4	B_5	2
O_{10}	C_1	C_2	C_3	C_4	C_5	3

In a real-life categorical data set, the partitions defined by attributes are not as perfect as that in the above example (i.e. the partitions defined by attributes are not always completely same); however, if the real classes are sufficiently distinguishable from each other, the objects in the same real classes have distinct value on some attributes from the objects in the other real classes, consequently, there exist some partitions defined by attributes which are similar to the real clustering of objects; at least, there exist some equivalence classes in these partitions which are similar to the real classes. Table 3.2 shows a simple example of such case. In Table 3.2, the partitions defined by attributes are not completely same, however, there are some equivalence classes (in the circles) in these partitions are as the same as the real classes. Our goal is to find such partitions and equivalence classes from the space of attributes partitions to construct the clustering of the objects.

Table 3.2: Example data set with six objects and three attributes

Objects	**Attribute 1**	**Attribute 2**	**Attribute 3**	**Real classes**
O_1	A_1	A_2	A_3	1
O_2	A_1	A_2	A_3	1
O_3	B_1	B_2	A_3	2
O_4	B_1	B_2	A_3	2
O_5	B_1	C_2	B_3	3
O_6	B_1	C_2	B_3	3

In addition, the number of attributes is usually less than the number of objects in a data set, thus it is possible to improve the clustering efficiency if the space of attributes partitions is employed for clustering.

3.3 MGR TECHNIQUE

3.3.1 Basic Idea of MGR

A good clustering of the objects should share as much information as possible with the partitions defined by each attribute (attributes partitions for short) (He et al., 2005a; He et al., 2008; Cristofor and Simovici, 2002; Deng et al., 2010). The aim of MGR algorithm is to search some equivalence classes from attributes partitions to form such a clustering of the objects that share as much information as possible with the attributes partitions. Concretely, MGR first selects a clustering attribute whose partition shares the most information with the partitions defined by other attributes, and then on the clustering attribute, the equivalence class with the highest intra-class similarity is output as a cluster, and the rest of the objects form the new current data set. Repeat the above two steps on the new current data set until all the objects are output. Figure 3.1 illustrates the basic steps of MGR technique.

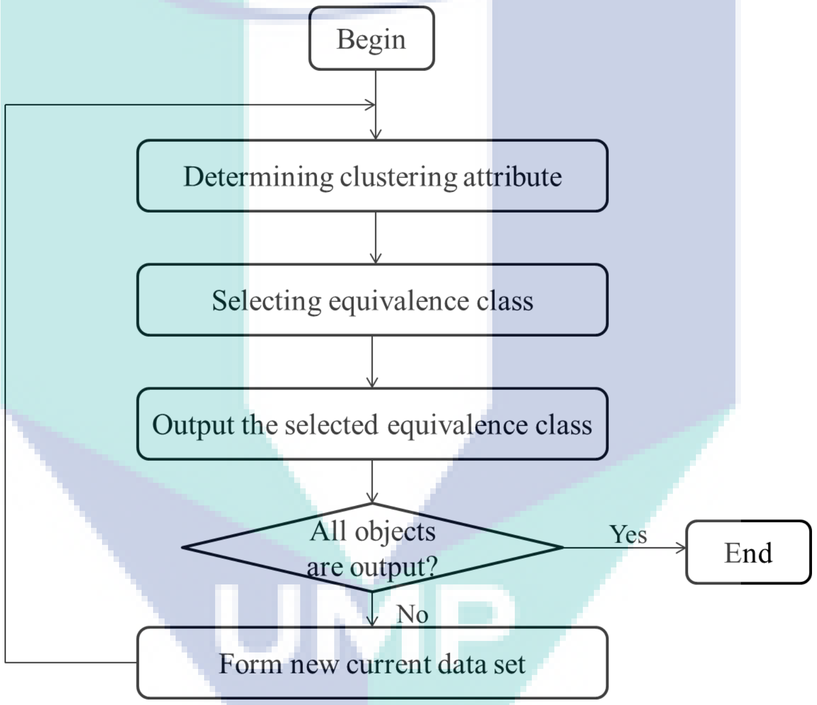

Begin
Determining clustering attribute
Selecting equivalence class
Output the selected equivalence class
All objects are output?
Yes
End
No
Form new current data set

Figure 3.1: The basic steps of MGR technique

- **Determining Clustering Attribute**

Two partitions share much information implies that the equivalence classes contained in these two partitions are similar to each other. For example, given six objects {1, 2, 3, 4, 5, 6} and three partitions on them, P_1 = {{1, 2}, {3, 4}, {5, 6}}, P_2 = {{1, 2}, {3, 4}, {5}, {6}}, P_3 = {{1}, {2, 3}, {4, 5, 6}}. It is obvious that P_1 and P_2

share more information than P_1 and P_3 or P_2 and P_3 because the equivalence classes contained in P_1 and P_2 are more similar (two equivalence classes are the same). That is, two partitions share much information means that they are similar (or close) to each other. Therefore, among the attributes partitions, the partition defined by the clustering attribute should be the most similar one to the partitions defined by all other attributes.

In decision tree classification algorithms C4.5(Quinlan, 1993), the information theory based concept of gain ratio is used as the similarity measure of the partition defined by an attribute with respect to the partition defined by class label attribute. In MGR algorithm, the definition of gain ratio is extended to mean gain ratio (MGR) to measure the similarity between the partition defined by an attribute and the partitions defined by all other attributes. In algorithms C4.5, the higher an attribute's gain ratio is, the more similar the attribute to the partition defined by class label attribute. Consequently, the higher an attribute's MGR is, the closer the partition defined by the attribute to the partitions defined by all other attributes. Thus, the attribute with the highest MGR is selected as the clustering attribute.

- **Selecting Equivalence Class**

Clusters of similar data objects have lower entropy than those of dissimilar ones (Barbara et al., 2002). In MGR algorithm, the entropy of cluster is used to select equivalence class from the partition defined by clustering attribute. The lower the entropy of a cluster is, the more similar the objects in the cluster. Thus, the equivalence class with the lowest entropy is selected as the splitting equivalence class and output as a cluster.

3.3.2 Information System

The data objects in a categorical data set are characterized by a set of categorical attributes. The concept of information system gives a formal description of objects in terms of their attribute values. Thus, a categorical data set can be formally described using an information system. The discussion about MGR in this chapter will be based on the notion of information system.

An information system is defined as a quadruple $S = (U, A, V, f)$, where:

- $U = \{x_1, x_2, ..., x_n\}$ is a set of *n* data objects, called a universe.
- $A = \{a_1, a_2, ..., a_m\}$ is a set of *m* attributes.
- $V = \bigcup_{j=1}^{m} V_{a_j}$, V_{a_j} is the domain of attribute a_j.
- $f{:}U{\times}A{\to}V$ is an information (knowledge) function such that $f(x_i, a_j) \in V_{a_j}$, for every $(x_i, a_j) \in U \times A$, $1 \le j \le m$ and $1 \le i \le n$.

An information system can be represented as an information table which consists of attribute-value pairs. Table 3.3 shows such an information table.

Table 3.3: An information system

U	a_1	a_2	$\cdots$	a_k	$\cdots$	a_m
x_1	$f(x_1, a_1)$	$f(x_1, a_2)$	$\cdots$	$f(x_1, a_k)$	$\cdots$	$f(x_1, a_m)$
x_2	$f(x_2, a_1)$	$f(x_2, a_2)$	$\cdots$	$f(x_2, a_k)$	$\cdots$	$f(x_2, a_m)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\ddots$	$\vdots$
x_n	$f(x_n, a_1)$	$f(x_n, a_2)$	$\cdots$	$f(x_n, a_k)$	$\cdots$	$f(x_n, a_m)$

The notion of information system can be regarded as a generalization of the concept of relational database table by labeling the rows of database table with the objects and labeling the columns with the attributes. Note that two different objects can have the same description, namely redundant tuples in an information table; However, in a relational database table it is not allowed.

3.3.3 Information Theory

Claude Shannon’s paper, “A Mathematical Theory of Communication”, published in the Bell System Technical Journal in 1948, is universally acknowledged as the beginning of information theory. This paper generated a profound influence on communication technology and attracted many researchers from a variety of backgrounds to contribute to the subject and expand it to many applications. Information theory has become an interdiscipline of electronic engineering, computer science, physics, statistics, and applied mathematics. It has been widely applied to numerous areas, such as communication networks, internet, cryptography, data compression, artificial intelligence, natural language processing, and so on.

The theoretical basis of Information theory includes probability theory and statistics. The most important concept in information theory is entropy, which is used to measure the amount of uncertainty related to a random variable. The higher the uncertainty is, the higher the entropy. Suppose X is a discrete random variable, D is the value set of X, and $P(x)$ is the probability of x given some $x \in D$, the entropy of X is defined as follows,

$$H(X) = -\sum_{x \in D} P(x) \log P(x) \tag{3.1}$$

There are two remarks about this definition. First, the base of the logarithmic is not specified. The base of the logarithmic determines the unit of information entropy. Base-2 unit called bit, is the most commonly used unit of information entropy. Other units include base-10 (common logarithm) and base-e (natural logarithm). Second, if $P(x)$ =0, the term $P(x)$ log $P(x)$ is defined to be 0.

For a categorical data set represented by an information system $S = (U, A, V, f)$, each attribute $a_i \in A$ can be regarded as a random variable, and each category of attribute a_i is regarded as a value of the random variable. Therefore, the above definition can be easily mapped onto a categorical data set. Let $U/a_i = \{X_1, \ldots, X_j, \ldots, X_h\}$ denote the partition on U defined by a_i where $X_j \subseteq U$ denotes an equivalence class (namely the block of objects which have the same value on a_i), and base-2 unit be the unit of information entropy, the Eq. (3.1) can be mapped onto a categorical data set as follows.

Definition 3.1. *Given attribute $a_i \in A$, suppose a_i defines partition $U/a_i = \{X_1, X_2, \ldots, X_h\}$, the entropy of a_i about the partition is defined as*

$$E(a_i) = -\sum_{s=1}^{h} P(X_s) \log_2 (P(X_s)) \tag{3.2}$$

where h is the domain size of a_i, $X_s \subseteq U$ is an equivalence class, $P(X_s) = \frac{|X_s|}{|U|}$, s =1, 2,…, h.

Another important concept in information theory is conditional entropy. Given two random variables X and Y, the conditional entropy of X given Y is used to measure the uncertainty about X after observing Y, and defined as follows.

$$H(X \mid Y) = -\sum_{y \in Y} P(y) \sum_{x \in X} P(x \mid y) \log P(x \mid y) \tag{3.3}$$

Replacing the random variables X and Y with attributes, Eq. (3.3) can be mapped onto a categorical data set as follows.

Definition 3.2. *Given attributes* $a_i, a_j \in A$, *suppose* a_i, a_j *define partitions* U/a_i ={X_1, X_2, …, X_h}, U/a_j ={Y_1, Y_2, …, Y_g}, *the conditional entropy (CE) of* a_i *with respect to* a_j *is defined as*

$$CE_{a_j}(a_i) = -\sum_{t=1}^{g} P(Y_t) \sum_{s=1}^{h} P(X_s|Y_t) \log_2 (P(X_s|Y_t)) \tag{3.4}$$

where $X_s, Y_t \subseteq U$, $P(Y_t) = \frac{|Y_t|}{|U|}$, $P(X_s|Y_t) = \frac{|Y_t \cap X_s|}{|Y_t|}$, s =1, 2, …, h and t =1, 2, ..., g.

Now that $H(X)$ stands for the information about X before Y is known, and $H(X|Y)$ stands for the information about X given Y, the difference $H(X)$-$H(X|Y)$ must represent the amount of information provided about X by Y. This quantity is called information gain, and defined as

$$I(X\,;\,Y) = H(X) - H(X \mid Y) \tag{3.5}$$

Similarly, Eq. (3.5) can be mapped onto a categorical data set as follows.

Definition 3.3. *Given attributes* $a_i, a_j \in A$*, the information gain (IG)of* a_i *with respect to* a_j *is defined as*

$$IG_{a_j}(a_i) = E(a_i) - CE_{a_j}(a_i) \tag{3.6}$$

Based on the concepts of entropy and information gain, algorithm C4.5 (Quinlan, 1993) defines the concept of gain ratio as follows.

Definition 3.4. *Given attributes* $a_i, a_j \in A$*, the gain ratio(GR)of* a_i *with respect to* a_j *is defined as*

$$GR_{a_j}(a_i) = \frac{IG_{a_j}(a_i)}{E(a_i)} \tag{3.7}$$

As mentioned in Section 3.3.1, MGR algorithm extends the definition of gain ratio to mean gain ratio to measure the similarity between the partition defined by an attribute and the partitions defined by all other attributes. It is defined as follows.

Definition 3.5. *Given attribute* $a_i \in A$*, the mean of gain ratio(MGR) of* a_i *is defined as*

$$MGR(a_i) = \frac{\sum_{j=1, j \neq i}^{|A|} GR_{a_j}(a_i)}{|A| - 1} \tag{3.8}$$

In addition, MGR algorithm uses the notion of entropy of cluster to select equivalence class from the partition defined by clustering attribute. Entropy of cluster is defined as follows.

Definition 3.6. *Assume the attributes in A are independent from each other, given a cluster* $C \subseteq U$*, the entropy of C is defined as*

$$Entropy(C) = E_C(a_1) + E_C(a_2) + \ldots + E_C(a_m) \tag{3.9}$$

where, $E_C(a_i)$, i=1, 2, …, m denotes the entropy of attribute a_i about the partition defined by a_i on C, which is calculated by Eq. (3.2).

3.4 MGR ALGORITHM

Figure 3.2 shows the MGR algorithm in details.

```
Algorithm:  MGR
Input:      $U$    //the set of objects
            $A$    //the set of attributes
            $k$    //the desired number of clusters
Output:     clustering of $U$

01 Begin
02     $Current\_Dataset = U$
03     $CNC = 1$.      // set current number of clusters
04     Repeat
05         for each attribute $a_i \in A$
06             $Calculate\_MGR(a_i)$
07         end for
08         // the attribute with the highest MGR is selected as clustering attribute
09         $a = Select\_attribute(A, \mathrm{MGR}(a_i))$
10         $p = U/a$     //get the partition defined by $a$
11         for each equivalence class $e_i$ in $p$
12             $Calculate\_Entropy(e_i)$     //using Eq. (3.9)
13         end for
14         // the equivalence class with the lowest entropy in $p$ is selected as the
15         // splitting equivalence class.
16         $e = Select\_equivalence\_class(p, \mathrm{Entropy}(e_i))$
17         $print(e)$     //output $e$ as a cluster
18         $CNC = CNC+1$
19         $Current\_Dataset = Current\_Dataset - e$
20     Until $CNC==k$ or $|Current\_Dataset /a_i| = =1$ for each attribute $a_i$
21     $print(Current\_Dataset)$   //output $Current\_Dataset$ as the last cluster
22 End
```

Figure 3.2: MGR algorithm

The function of $Calculate\_MGR(a_i)$ is implemented by using Eq. (3.2), Eq. (3.4), Eq. (3.6), Eq. (3.7), and Eq. (3.8), the details are described in Figure 3.3.

```
01 Begin
02     Determine equivalence classes in the partition $Current\_Dataset / a_i$
03     Calculate $E(a_i)$
04     for each attribute $a_j \in A\,(j \neq i)$
05         Calculate $CE_{a_j}(a_i)$
06         Calculate $IG_{a_j}(a_i)$
07         Calculate $GR_{a_j}(a_i)$
08     end for
09     Calculate $MGR(a_i)$
10 End
```

Figure 3.3: The procedure of calculating MGR (a_i)

In view of the size of some selected equivalence classes might be very small, they are regarded as outlier. If the size of the equivalence class with the lowest entropy is less than a specified *threshold*, the equivalence class with the next lowest entropy will be checked until the size of an equivalence class is greater than the threshold. Thus, the procedure of selecting the equivalence class (Line 16 in Figure 3.2) is fined as is shown in Figure 3.4.

```
01 Begin
02     Size_flag = true
03     Repeat
04         e = Select_equivalence_class (p, Entropy($e_i$))
05         if |e| < threshold
06             Size_flag = false
07             p = p-e
08         else
09             Size_flag = true
10     Until Size_flag==true
11 End
```

Figure 3.4: The procedure of selecting the equivalence class

It is possible that the sizes of all first $|V(a)\text{-}1|$ equivalence classes are less than the *threshold*. In that case, the clustering attribute *a* is gave up and the attribute with the next highest MGR will be checked.

There is a remark about the termination of MGR algorithm. It is not necessarily to specify the number of clusters *k* for the algorithm. If the number of clusters is not specified, the algorithm will terminate as each attribute in the current dataset only has one equivalence class. This is a more natural way than specifying the number of clusters especially when user experiences difficulties in specifying the number of clusters.

Next, an illustrative example of the MGR algorithm is presented.

Example 3.1. Table 3.4 shows a data set of students' enrollment qualification used to illustrate the application of the MGR algorithm. There are eight objects with seven categorical attributes. The number of clusters is set to 3.

Table 3.4: An information system of student's enrollment qualification

Student	Degree	English	Experience	IT	Mathematics	Programming	Statistics
1	Ph.D	good	medium	good	good	good	good
2	Ph.D	medium	medium	good	good	good	good
3	M.Sc	medium	medium	medium	good	good	good
4	M.Sc	medium	medium	medium	good	good	medium
5	M.Sc	medium	medium	medium	medium	medium	medium
6	M.Sc	medium	medium	medium	medium	medium	medium
7	B.Sc	medium	good	good	medium	medium	medium
8	B.Sc	bad	good	good	medium	medium	good

Source: Herawan et al. (2010)

First, the mean of gain ratio of each attribute is calculated. Let us take attribute "Degree" as an example. Following the algorithm shown in Figure 3.3, the mean gain ratio of attribute "Degree" is calculated as follows.

Attribute "Degree" defines the partition

$$U/\text{ Degree} = \{\{1, 2\}, \{3, 4, 5, 6\}, \{7, 8\}\}.$$

Using Eq. (3.2), the entropy of attribute "Degree" is calculated as follows

$$E(\text{Degree}) = -\frac{1}{4}\log_2{\frac{1}{4}} - \frac{1}{2}\log_2{\frac{1}{2}} - \frac{1}{4}\log_2{\frac{1}{4}} = 1.5$$

Next, for each attribute a_j ($a_j \neq a_i$ = Degree), $CE_{a_j}(a_i)$, $IG_{a_j}(a_i)$, and $GR_{a_j}(a_i)$ are calculated, respectively. Let us take attribute "English" as an example.

Attribute "English" defines the partition

$$U/\text{ English} = \{\{1\}, \{2, 3, 4, 5, 6, 7\}, \{8\}\}.$$

Using Eq. (3.4), the conditional entropy of attribute "Degree" with respect to attribute "English" is calculated as follows

$$CE_{\text{English}}(\text{Degree}) = -(\frac{1}{8}\times 0 + \frac{3}{4}(\frac{1}{6}\log_2{\frac{1}{6}} + \frac{2}{3}\log_2{\frac{2}{3}} + \frac{1}{6}\log_2{\frac{1}{6}}) + \frac{1}{8}\times 0)$$

$$= 0.939$$

Using Eq. (3.6), the information gain of attribute "Degree" with respect to attribute "English" is calculated as follows

$$IG_{\text{English}}(\text{Degree}) = 1.5 - 0.939 = 0.561$$

Finally, the gain ratio of attribute "Degree" with respect to attribute "English" is obtained by using Eq. (3.7)

$$GR_{\text{English}}(\text{Degree}) = \frac{0.561}{1.5} = 0.374.$$

With the same process, the gain ratios of attribute "Degree" with respect to other attributes are obtained,

$GR_{\text{Experience}}(\text{Degree}) = 0.541$,

$GR_{\text{IT}}(\text{Degree}) = 0.667$,

$GR_{\text{Mathematics}}(\text{Degree}) = 0.333$,

$GR_{\text{Programming}}(\text{Degree}) = 0.333$,

$GR_{\text{Statistics}}(\text{Degree}) = 0.230$.

Consequently, according to Eq. (3.8), the mean gain ratio of attribute "Degree" is obtained,

$$MGR(\text{Degree}) = \frac{0.374 + 0.541 + 0.667 + 0.333 + 0.333 + 0.23}{6} = 0.413.$$

Following the same procedure, the mean gain ratios of other attributes are calculated. The results of gain ratio and mean gain ratio are summarized in Table 3.5.

Table 3.5: GR and MGR of all attributes in Table 3.4

Attribute (with respect to)	**Degree**	**English**	**Experience**	**IT**	**Math**	**Programming**	**Statistics**	**MGR**
Degree	-	0.374	0.541	0.667	0.333	0.333	0.230	0.413
English	0.529	-	0.305	0.293	0.236	0.236	0.293	0.315
Experience	1.000	0.399	-	0.384	0.384	0.384	0.000	0.425
IT	1.000	0.311	0.311	-	0.000	0.000	0.189	0.302
Mathematics	0.500	0.250	0.311	0.000	-	1.000	0.189	0.375
Programming	0.500	0.250	0.311	0.000	1.000	-	0.189	0.375
Statistics	0.344	0.311	0.000	0.189	0.189	0.189	-	0.204

Second, the clustering attribute with the highest MGR is chose. Table 3.5 shows that attribute “Experience” has the highest MGR, thereby, it is chose as clustering attribute.

Third, the splitting equivalence class with the minimum entropy is determined. Attribute “Experience” defines a partition {{1, 2, 3, 4, 5, 6}, {7, 8}}. C_1 and C_2 are used to denote {1, 2, 3, 4, 5, 6} and {7, 8}, respectively. According to the Definition 3.6, the entropies are calculated as follows

$$\begin{aligned}\text{Entropy}(C_1) &= E_{C_1}(\text{Degree}) + E_{C_1}(\text{English}) + E_{C_1}(\text{Experience}) + E_{C_1}(\text{IT}) \\ &\quad + E_{C_1}(\text{Math}) + E_{C_1}(\text{Programming}) + E_{C_1}(\text{Statistics}) \\ &= (-\frac{1}{3}\log_2{\tfrac{1}{3}} - \frac{2}{3}\log_2{\tfrac{2}{3}}) + (-\frac{1}{6}\log_2{\tfrac{1}{6}} - \frac{5}{6}\log_2{\tfrac{5}{6}}) + 0 \\ &\quad + (-\frac{1}{3}\log_2{\tfrac{1}{3}} - \frac{2}{3}\log_2{\tfrac{2}{3}}) + (-\frac{1}{3}\log_2{\tfrac{1}{3}} - \frac{2}{3}\log_2{\tfrac{2}{3}}) \\ &\quad + (-\frac{1}{3}\log_2{\tfrac{1}{3}} - \frac{2}{3}\log_2{\tfrac{2}{3}}) + 1 \\ &= 5.323\end{aligned}$$

$$\begin{aligned}\text{Entropy}(C_2) &= E_{C_2}(\text{Degree}) + E_{C_2}(\text{English}) + E_{C_2}(\text{Experience}) + E_{C_2}(\text{IT}) \\ &\quad + E_{C_2}(\text{Math}) + E_{C_2}(\text{Programming}) + E_{C_2}(\text{Statistics}) \\ &= 0 + 1 + 0 + 0 + 0 + 0 + 1 \\ &= 2\end{aligned}$$

In this example, the *threshold* of the size of splitting equivalence class is set to be 3% of the number of data set, that is, *threshold* = 0.03*8 = 0.24 < 1. Obviously, set {7, 8} has the lowest entropy and its size is greater than threshold. Hence, set {7, 8} is selected as splitting equivalence class.

Finally, splitting equivalence class {7, 8} is output as a cluster, equivalence class {1, 2, 3, 4, 5, 6} is regarded as the new current dataset to further process.

Over the new current dataset, the above procedure is repeated. This time, attribute "Degree" is chose as clustering attribute and {1, 2, 3, 4, 5, 6} is partitioned to two clusters {1, 2} and {3, 4, 5, 6}. Subsequently, set {1, 2} is selected as the splitting equivalence class since it has the lowest entropy, and output as a cluster. Set {3, 4, 5, 6} becomes the new current data set. Since the current number of clusters reaches 3 so far, the procedure of iteration stops. The current dataset is output as the final cluster. In the end, the data set is partitioned to three clusters, C_1 = {7, 8}, C_2 = {1, 2}, C_3 = {3, 4, 5, 6}.

3.5 MGR COMPUTATIONAL COMPLEXITY

Given a data set, assume n is the number of objects, m is the number of attributes, l is the maximum number of values in the attribute domains and k is the required number of clusters. To achieve k clusters, the algorithm has to runs k-1iterations. In each iteration, the time to determine equivalence classes for each attribute is mn, the time to compute the entropy of attributes is ml, the time to calculate the conditional entropy is m^2l, the time to calculate the IG and GR is $2m^2$, the time to calculate MGR is m, the time to determine the clustering attribute is m, the time to compute the entropy of the equivalence classes on the clustering attribute is ml^2. The whole time for k-1iterations is $km^2(2+l)+km(n+l+l^2+2)$. Generally, $l \ll n$, thus, the computational complexity is $O(km^2l + kmn)$, which is polynomial time.

3.6 COMPARISONS WITH MMR

3.6.1 Limitations of MMR

In this section, the limitations of MMR algorithm are analyzed and illustrated with some examples.

i. *MMR algorithm is biased toward the attribute with the smallest value domain size or with the most unbalanced partition as determining the partitioning attribute (Step 4 in Figure 2.7).*

There are two reasons for this limitation. First, MMR has two properties as follows.

Proposition 3.1. *Given an information system S = (U, A, V, f), if an attribute defines an one-equivalence-class partition, then the attribute has minimum Min-Roughness, i.e. MMR.*

Proof. Suppose attribute $a_i \in A$ defines a one-equivalence-class partition, which means all the objects in U have the same value on attribute a_i. Suppose the same value is γ, for any attribute $a_j \in A(j \neq i)$, the lower approximation and upper approximation of $T(a_i = \gamma)$ with regard to a_j are calculated as follows

$$\left|\underline{T}_{a_j}(a_i = \gamma)\right| = \left|\overline{T}_{a_j}(a_i = \gamma)\right| = |U|$$

Then, using Eq. (2.31), the roughness of $T(a_i = \gamma)$ with regard to a_j is obtained

$$R_{a_j}(T|a_i = \gamma) = 1 - \frac{\left|\underline{T}_{a_j}(a_i = \gamma)\right|}{\left|\overline{T}_{a_j}(a_i = \gamma)\right|} = 0$$

Using Eq. (2.32) and Eq. (2.33), the mean roughness and Min-Roughness of attribute a_i are obtained

$$Rough_{a_j}(a_i) = 0\text{, and } MR(a_i) = 0\,.$$

From the definitions in Section 2.3.8, it is easy to get that $MR(a) \geq 0$ for each $a \in A$. Hence, attribute a_i has minimum Min-Roughness, i.e. MMR. □

Proposition 3.2. *Given an information system S = (U, A, V, f), if an attribute defines a partition with one-element equivalence classes , then the attribute has maximum Min-Roughness in A.*

Proof. Suppose attribute $a_i \in A$ defines a partition with one-element equivalence classes, which means each object in U has a different value on attribute a_i.

For any attribute $a_j \in A$ ($j \neq i$), suppose there are N_{j_1} one-element equivalence classes, N_{j_2} non one-element equivalence classes in the partition U/a_j. Using the Eq. (2.31), the following two conclusions are obtained.

(1) There are N_{j_1} equivalence classes in the partition U/a_i whose roughness equal 0 with respect to attribute a_j.

(2) There are $|U| - N_{j_1}$ equivalence classes in the partition U/a_i whose roughness equal 1 with respect to attribute a_j.

Thereby, using Eq. (2.32), the mean roughness on attributes a_i with respect to a_j is calculated as follows

$$Rough_{a_j}(a_i) = \frac{|U| - N_{j_1}}{|U|}$$

Conversely, the roughness of all equivalence classes in the partition U/a_j with respect to attribute a_i equal 0, hence the mean roughness on attributes a_j with respect to a_i equal 0, that is

$$Rough_{a_i}(a_j) = 0$$

Finally, the following formula is obtained by using Eq. (2.33).

$$MR(a_i) = \frac{\min\{|U| - N_{j_1}, j = 1,\ldots,m, j \neq i\}}{|U|} \geq MR(a_j) = 0$$

$MR(a_i) = MR(a_j) = 0$ when there exists an attribute a_j with $|U| - N_{j_1} = 0$, which means attribute a_j also defines a partition with one-element equivalence classes. Hence, if an attribute defines a partition with one-element equivalence classes, then the attribute has maximum Min-Roughness in *A*. □

In the following discussion, the attribute that defines a one-equivalence-class partition is called P_1-type attribute, the attribute defines a partition with one-element equivalence classes is called P_2-type attribute. Further, let us give a deep insight into these two types attributes. A P_1-type attribute has two properties: (1) it has the smallest value domain size; (2) it has the most unbalanced partition. Reversely, for a P_2-type attribute, (1) it has the biggest value domain size; (2) it has the most balanced partition.

The two propositions imply that the attribute with the smaller value domain size or with the more unbalanced partition usually has lower Min-Roughness, which means MMR algorithm prefers to select such attribute as the partition attribute. For example, if a partitioning attribute is required to be selected (Step 4 in Figure 2.7) from an attribute set which includes a P_1-type attribute, according to MMR algorithm, the P_1-type attribute will be selected since it has MMR value, while it is impossible to be selected for a P_2-type attribute because of its maximum MR value. This finally results in the extreme selection of MMR algorithm as determining the partitioning attribute, namely, MMR algorithm is biased toward the attribute with the smallest value domain size or with the most unbalanced partition (although P_1-type attribute is excluded when the algorithm is implemented).

Second, from the definition of roughness (Eq.(2.31)), it can be seen that the formula only focuses on the precision of *X* with respect to a_j, regardless of the size of *X* and the distribution of attribute a_i. This also contributes to the extreme selection as determining the partitioning attribute.

Determining partitioning attribute is the key of MMR algorithm. Such extreme selections will decrease the clustering accuracy of MMR algorithm; after all, the real clusters are not always embedded in such attributes. The following examples illustrate these two cases.

Example 3.2. There are five objects in a small data set as is shown in Table 3.6. Each object has four categorical attributes: Size, Material, Shape, and Colour. The domain sizes of these four attributes are 2, 3, 4, and 5, respectively.

Table 3.6: Example data set with five objects and four attributes

U	**Size**	**Material**	**Shape**	**Colour**
1	Big	Wood	Circle	Red
2	Small	Plastic	Square	Green
3	Big	Wood	Ellipse	Yellow
4	Small	Plastic	Circle	White
5	Small	Metal	Triangle	Black

Following MMR algorithm, attribute “Size”, which has the smallest domain size, is selected to split the dataset in the first selection. Suppose three clusters are required, attribute “Material” will be selected in the second selection (iteration) to split objects set {2, 4, 5}. Note that attribute “Material” also has the smallest domain size except for single-value attribute “Size” (excluded) when objects set is {2, 4, 5}.

Example 3.3. Using a data generator (Cristofor et al., 2002), a data set of 5 objects with 5 attributes a_1, a_2, ..., a_5 is generated as is shown in Table 3.7.

Table 3.7: Example data set with five objects and five attributes

U	a_1	a_2	a_3	a_4	a_5
1	0	0	1	0	0
2	0	1	1	1	1
3	1	1	0	1	1
4	0	1	0	0	0
5	0	0	0	1	0

The data generator sets value 0 on most attributes (four out of five) for objects 1, 4, and 5, and sets value 1 on most attributes (four out of five) for objects 2 and 3, that is, the reference partition is {0, 1, 1, 0, 0}. The real clusters can be obtained from this reference partition, namely {{1, 4, 5}, {2, 3}}. Following MMR algorithm, attribute a_1 is selected as the partition attribute. Thereby, the clustering result is {{3}, {1, 2, 4, 5}}.

Note that, MMR algorithm selects the attribute with the most unbalanced partitions among all attributes in this example. In the above dataset, each attribute has the same number of equivalence classes, i.e. 2. The numbers of the objects in each equivalence class are listed in Table 3.8. Values 0 and 1 in Table 3.7 denote different categories of an attribute regardless of order, therefore, class 1 in Table 3.8 denotes the equivalence class that includes object 1, and class 2 denotes the other equivalence class. It is obvious that attribute a_1 has the most unbalanced partitions.

Table 3.8: The numbers of the objects in each equivalence class

Equivalence classes	a_1	a_2	a_3	a_4	a_5
class 1	4	2	2	2	3
class 2	1	3	3	3	2

ii. *Selecting the current longest cluster to further binary split (Step 7 in Figure 2.7) is not always consistent with the natural distribution of clusters.*

For unsupervised learning, the length of the clusters is not known in advance. There exist some clusters with longer length in data sets. Therefore, using the length of clusters as the criterion is not natural, that is, it is not always consistent with the natural distribution of clusters. The following example illustrates the view.

Example 3.4. Table 3.9 shows an animal world data set which is modified from (Hu, 1995). There are seven objects with seven categorical attributes: Hair, Teeth, Feet, Eat, Milk, Fly, and Swim. The attribute "Type" shows the real classes of the animals, i.e. {mammal, bird, reptile}. Note, attribute "Type" does not participate in the process of clustering.

Table 3.9: The animal dataset

Animal	**Hair**	**Teeth**	**Feet**	**Eat**	**Milk**	**Fly**	**Swim**	**Type**
Tiger	Y	pointed	claw	meat	Y	N	Y	mammal
Cheetah	Y	pointed	claw	meat	Y	N	Y	mammal
Giraffe	Y	blunt	hoof	grass	Y	N	N	mammal
Zebra	Y	blunt	hoof	grass	Y	N	N	mammal
Albatross	N	N	claw	grain	N	Y	Y	bird
Eagle	N	N	claw	meat	N	Y	N	bird
Viper	N	pointed	N	meat	N	N	N	reptile

Following MMR algorithm, first, attribute "Hair" is selected to split the dataset. With attribute "Hair", two clusters are obtained, i.e. P_1 = {Tiger, Cheetah, Giraffe, Zebra} and P_2 = {Albatross, Eagle, Viper}. Next, cluster P_1 is selected to be further split because P_1 is the current longest node among all the clusters. This time, attribute "Teeth" is chose as partition attribute. With attribute "Teeth", two clusters are obtained, i.e. P_{11} = {Tiger, Cheetah} and P_{12} = {Giraffe, Zebra}. Finally, three clusters are obtained, C_1 = {Tiger, Cheetah}, C_2 = {Giraffe, Zebra}, and C_3 = {Albatross, Eagle, Viper}. However, in terms of attribute "Type", the real clusters are C_1 = {Tiger, Cheetah, Giraffe, Zebra}, C_2 = {Albatross, Eagle}, and C_3 = {Viper}.

Assuming partition P_2 was selected to be further processed instead of the longest node P_1 after splitting by attribute "Hair", obviously, the clustering results will be the same as the real clusters. Therefore, selecting the longest node is not always consistent with the natural distribution of clusters.

iii. *MMR algorithm has to store all clusters produced during the execution of program*

In order to select the current longest clusters for the further process, MMR algorithm has to store all clusters produced during the execution of program, which is space consuming. All clusters correspond to all objects, suppose there are n objects in dataset and only store the ID of the objects in clusters, then at least n space nodes are required to store them, that is, the extra space complexity is $O(n)$.

3.6.2 Comparison Between MGR and MMR

MGR algorithm can overcome the above limitations of MMR algorithm well. Corresponding to those limitations, MGR algorithm has the following advantages:

i. *MGR algorithm is not biased toward extreme selections.*

There are two reasons for this view. First, it can not be confirmed that P_1-type and P_2-type attributes have the maximum or minimum of mean gain ratio. Thus, they are not necessarily selected as clustering attribute.

Second, in the decision tree learning algorithm C4.5, the reason for using gain ratio measure is to avoid extreme selection caused by information gain measure. The mean gain ratio used in MGR algorithm has the same principle, thus it can avoid extreme selections.

Example 3.5. Let us reconsider the clustering of the data set in Example 3.3. Here the data set is reclustered using MGR algorithm.

Following MGR algorithm, the results of gain ratio and mean gain ratio are summarized in Table 3.10.

Table 3.10: Gain Ratio of all attributes in Table 3.7

Attribute (with respect to)	a_1	a_2	a_3	a_4	a_5	**MGR**
a_1	-	0.237	0.237	0.237	0.446	0.289
a_2	0.176	-	0.021	0.021	0.433	0.162
a_3	0.176	0.021	-	0.021	0.021	0.059
a_4	0.176	0.021	0.021	-	0.433	0.162
a_5	0.332	0.433	0.021	0.433	-	0.304

Table 3.10 shows that attribute a_5 has the highest MGR. Therefore, according to MGR algorithm, attribute a_5 is chosen as the clustering attribute. Attribute a_5 partitions the data set shown in Table 3.7 into two clusters, that is {{1, 4, 5}, {2, 3}}. It has been mentioned in Example 3.3 that the data generator set the real clusters as {{1, 4, 5}, {2, 3}}. Thereby, the clustering result generated by MGR algorithm on this data set is the same as the real clusters.

ii. *In each iteration, MGR algorithm output the cluster found regardless of its length and perform binary split on the remaining objects.*

The way of MGR algorithm is more natural than that of MMR algorithm, that is, it is more consistent with the real distribution of clusters. The following example illustrates the view.

Example 3.6. Let us reconsider the clustering of the data set in Example 3.4. Here the data set is reclustered using MGR algorithm.

Following MGR algorithm, two iterations are needed to obtain three clusters from the data set. The results of gain ratio and mean gain ratio in the first iteration are summarized in Table 3.11 which shows the one-to-one relationship between attributes by using the same attributes for both axes. Each cell (except for the cells in the last column) of Table 3.11 gives the gain ratio value of an attribute with respect to another attribute and the last column gives the MGR value of each attribute.

Table 3.11: Gain Ratio of all attributes in Table 3.9 in the first iteration

Attribute (with respect to)	**Hair**	**Teeth**	**Feet**	**Eat**	**Milk**	**Fly**	**Swim**	**MGR**
Hair	-	0.601	0.420	0.420	1.000	0.477	0.021	0.490
Teeth	0.380	-	0.633	0.702	0.380	0.554	0.197	0.474
Feet	0.300	0.715	-	0.664	0.300	0.212	0.378	0.428
Eat	0.300	0.793	0.664	-	0.300	0.290	0.300	0.441
Milk	1.000	0.601	0.420	0.420	-	0.477	0.021	0.490
Fly	0.544	1.000	0.338	0.463	0.544	-	0.007	0.483
Swim	0.021	0.311	0.529	0.420	0.021	0.006	-	0.218

Attribute “Hair” has the highest MGR among all the attributes, thus it is chosen as clustering attribute. Two clusters are obtained by attribute “Hair”, i.e. T_1 = {Tiger, Cheetah, Giraffe, Zebra} and T_2 = {Albatross, Eagle, Viper}. Next, cluster T_1 is output since it has the lowest entropy and cluster T_2 is to be further split in the second iteration. The results of gain ratio and mean gain ratio in the second iteration are summarized in Table 3.12.

Table 3.12: Gain Ratio of all attributes in Table 3.9 in the second iteration

Attribute (with respect to)	**Hair**	**Teeth**	**Feet**	**Eat**	**Milk**	**Fly**	**Swim**	**MGR**
Hair	-	0	0	0	0	0	0	0
Teeth	0	-	1	0.274	0	1	0.274	0.425
Feet	0	1	-	0.274	0	1	0.274	0.425
Eat	0	0.274	0.274	-	0	0.274	1	0.304
Milk	0	0	0	0	-	0	0	0
Fly	0	1	1	0.274	0	-	0.274	0.425
Swim	0	0.274	0.274	1	0	0.274	-	0.304

In the second iteration, attribute “Teeth” is the first attribute that has the highest MGR. Therefore, it is selected as clustering attribute. Two clusters are obtained by attribute “Teeth”, i.e. T_{21} = {Albatross, Eagle} and T_{22} = {Viper}. Finally, three clusters are obtained, i.e. C_1 = {Tiger, Cheetah, Giraffe, Zebra}, C_2 = {Albatross, Eagle}, and C_3 = {Viper}, which is the same as the real clusters.

The example illustrates that the way of MGR algorithm is more natural than that of MMR algorithm.

iii. *For MGR algorithm, there is no need of storing all clusters produced during the execution of program.*

In each iteration, MGR first selects a clustering attribute, and then selects an equivalence class on the clustering attribute, finally outputs the seleted equivalence class as a cluster (Line 17 in Figure 3.2). In other words, all the clusters produced in the iterations are directly printed rather than stored in memory, thus, there is no need of extra space for storing all clusters during the execution of program.

3.7 SUMMARY

This chapter reveals the significance of attributes for categorical data clustering and proposes a novel attribute-oriented hierarchical divisive clustering algorithm named MGR for categorical data. Mean gain ratio and entropy of clusters, two information theory based concepts, are introduced in the implementation of MGR algorithm. From the analysis of the limitations of MMR algorithm and the comparison between MGR and MMR, it can be seen that MGR can overcome the limitations of MMR algorithm. The polynomial computational complexity implies that MGR still maintains high efficiency.

CHAPTER 4

IG-ANMI: AN IMPROVED GENETIC CLUSTERING ALGORITHM FOR CATEGORICAL DATA

4.1 INTRODUCTION

Deng et al. (2010) propose G-ANMI, which is an ANMI based genetic clustering algorithm for categorical data. It has been demonstrated that G-ANMI algorithm is superior or comparable to existing clustering algorithms for categorical data including ALG-RAND, ccdByEnsemble, *k*-ANMI, *k*-modes, TCSOM, and Squeezer, according to clustering accuracy. However, the low efficiency of G-ANMI is a considerable obstacle before it can be widely used in practice. The low efficiency of the algorithm is mainly contributed by genetic algorithm in which lots of iterations are needed to find globally optimal solution (Deng et al., 2010). Especially when a big population size is used, each iteration will take much time. Hence, new methods which can reduce the number of iterations of genetic algorithm in G-ANMI are desired.

G-ANMI algorithm first randomly generates a set of partitions of objects which form a population. These randomly generated partitions are far from the distribution of the real classes in the processed data set. The farther these partitions are from the distribution of the real classes, the more iteration G-ANMI needs to reach the optimal solution. Thus, improve the initial population is a possible method to reduce the number of iteration of G-ANMI. As described in the last chapter, MGR algorithm implements clustering from the viewpoint of the attribute, in which the partitions defined by attributes and the equivalence classes in these partitions are used to build the clustering of the objects. Inspired by the idea of MGR, the space of attributes partitions are tried to be used to construct the initial population of G-ANMI. In this chapter, an improved genetic clustering algorithm for categorical data is proposed, termed IG-ANMI. Based on G-ANMI, IG-ANMI improves the method of initialization. Prior to the description of the algorithm, the preliminary knowledge about genetic algorithms (GAs) and algorithm G-ANMI are introduced. The chapter is structured as follows:

- Section 4.2 introduces GAs.
- Section 4.3 describes the algorithm G-ANMI.
- Section 4.4 describes the algorithm IG-ANMI.

4.2 GENETIC ALGORITHMS

GAs (Holland, 1992; Mitchell, 1998; Man et al., 2001) are a part of evolutionary computing, which is a rapidly growing area of artificial intelligence. GAs were invented by Holland (1992) and developed by him and his students and colleagues.

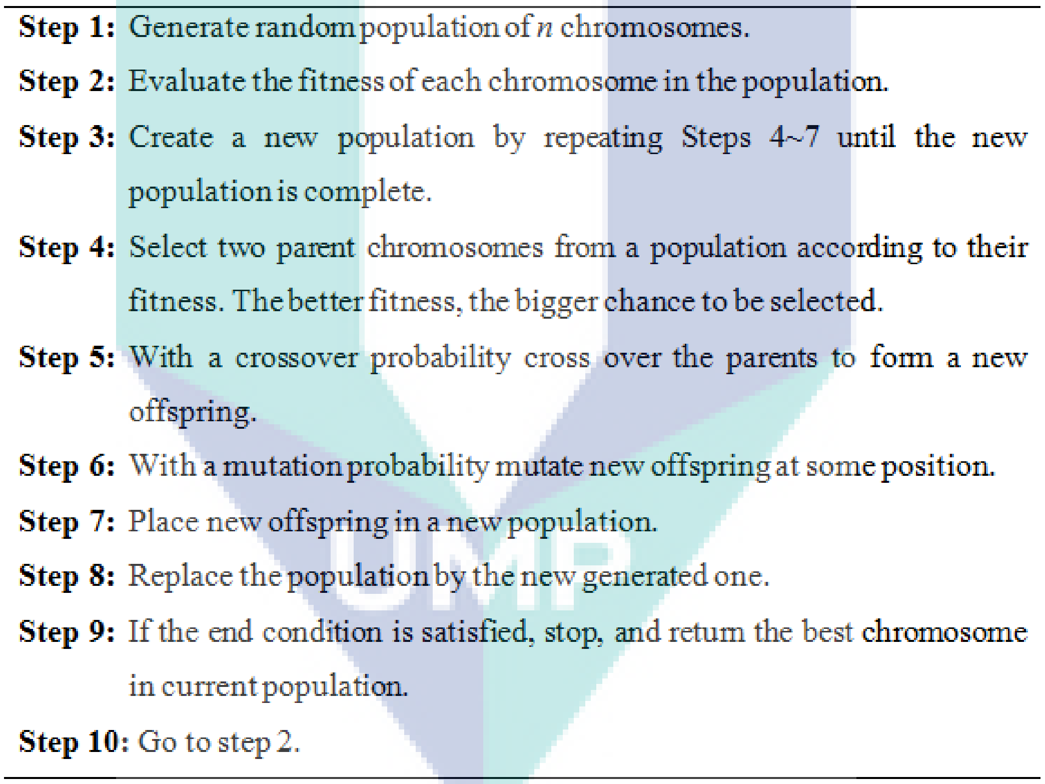
4.2.1 Biological Background

Cell is the basic component of an organism. In the nucleus of each cell there is a set of threadlike linear strand of DNA called chromosomes. Chromosomes carry the hereditary information which are encoded and stored in the hereditary units called genes. Each gene has a particular location in a chromosome and represents a particular characteristic of an organism, such as the number of legs, intelligence and so on. Each characteristic has some different settings, for instance, the number of leges may be 2, 4, 6, etc. All the hereditary information carried by chromosomes in a cell's nucleus is referred to genome. Specific set of genes of an organism and their settings are called the organism's genotype.

When sexual reproduction takes place between two organisms, two chromosomes from parent will recombine by performing crossover to produce two new chromosomes. There might exsit some mistakes during cell's reproduction, that is, some genes might be mutated. It is possible that the mutated genes express completely new characteristics. These basic processes including natural selction, mutation and recombination enable the life on earth continuously evolve.

4.2.2 The Outline of Basic GAs

The first step of GAs is to generate a set of chromosomes called initial population, which are generated randomly in most cases. Each chromosome in current population is then evaluated and assigned a fitness value. Based on these fitness values, some chromosomes in current population are chose and used to generate new population. In general, the higher the fitness value a chromosome has, the more chances it has to be selected. Repeat the above steps on the new generated population until the best fitness vaule or the number of iterations exceeds the predefined threshold. Figure 4.1 shows the steps of basic GAs.



Step 1: Generate random population of *n* chromosomes.

Step 2: Evaluate the fitness of each chromosome in the population.

Step 3: Create a new population by repeating Steps 4~7 until the new population is complete.

Step 4: Select two parent chromosomes from a population according to their fitness. The better fitness, the bigger chance to be selected.

Step 5: With a crossover probability cross over the parents to form a new offspring.

Step 6: With a mutation probability mutate new offspring at some position.

Step 7: Place new offspring in a new population.

Step 8: Replace the population by the new generated one.

Step 9: If the end condition is satisfied, stop, and return the best chromosome in current population.

Step 10: Go to step 2.

Figure 4.1: The outline of basic GAs

When the Basic GAs is used in practice, the implementation of some steps varies with the different applications, including encoding of chromosomes, how to select parent chromosomes, and what types of crossover and mutation are used. These problems will be discussed in the following sections, respectively.

4.2.3 Encoding of Chromosomes

When GA is being used to solve a problem, the possible solutions of this problem have to be encoded as chromosomes. The encoding varies with the problems. There are several successfully used ways of encoding for a chromosome, such as binary encoding, permutation encoding, value encoding, and tree encoding. The mostly used binary encoding and value encoding are introduced below.

- Binary Encoding

  Each chromosome is encoded as a strand of 1 or 0. Table 4.1 shows the example of chromosomes with binary encoding.

Table 4.1: The example of chromosomes with binary encoding

Chromosome A	1011001011001010111100101
Chromosome B	1111111000001100000111111

- Value Encoding

  It is possibly difficult using binary encoding to solve some special problems in which some complex values are involved. Each chromosome in value encoding is encoded as a strand of some values. Values depend on the problem, they might be characters, real numbers, words, or some complex objects. Such encoding method is suitable to solve some particular problems; however, new mutation and crossover methods special for these problems are usually required. Table 4.2 shows the example of chromosomes with value encoding.

Table 4.2: The example of chromosomes with value encoding

Chromosome A	white, yellow, red, pink, blue, green
Chromosome B	3.19 8.45 2.37 6.23 9.77 11.55
Chromosome C	UYTORSVNYPLKWAHJBQWIMS

4.2.4 Fitness Function

During the evolution of population, each chromosome has to be evaluated how fit it is to be a solution of the problem. Such evaluation is performed by the fitness function. The return value of fitness function named fitness value usually is a positive number. The higher the fitness value is, the fitter the chromosome to the solved problem (Man et al., 2001).

The fitness function is usally defined by transforming the objective function. For instance, $f_i / \overline{f}$ is one of commonly used definitions of fitness function, where f_i denotes the evaluation of i^{th} chromosome and can be calculated by using objective function, and $\overline{f}$ denotes the average evaluation of all the chromosomes. Moreover, a chromosome's rank in the population can also be used to compute the fitness value.

4.2.5 Selection Operators

Selection refers to selecting some chromosomes in the population to create new offspring according to chromosomes' fitness. In GA, the selection operation complies with Darwin's evolution theory, that is, the fittest ones should be selected. There are a number of ways to do selection, such as elitism, roulette wheel selection, rank selection, tournament selection, and Boltzman selection (Mitchell, 1998). The most commonly used methods including roulette wheel selection and elitism are introduced below.

i. Roulette Wheel Selection

In this method, the chromosomes are mapped onto a roulette wheel, where each chromosome occupies a piece of space whose area is proportion to its fitness value. Figure 4.2 shows an example of such mapping.

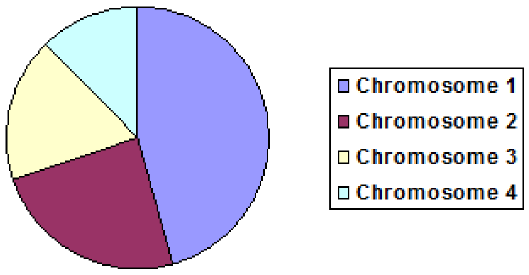


Figure 4.2: The mapping of chromosomes onto a roulette wheel

Imaging a die is thrown on the roulette wheel; the chromosome which finally contains the die will be selected. Obviously, the bigger the area of a chromosome is, the more times it is selected. The algorithm shown in Figure 4.3 simulates the process.

```
Step 1:   Calculate the sum $M$ of fitness of all chromosomes in population.
Step 2:   Randomly generate a number $v$ in interval [0, $M$].
Step 3:   Calculate the sum $s$ of fitness from chromosome 1. When the sum
          $s$ is greater than $v$, stop and return the chromosome where you are.
```

Figure 4.3: The algorithm of roulette wheel selection

ii. Elitism

Elitism guarantees some best first chromosomes survive in the new population by directly copying them to new population. Elitism is usually followed by other selection methods which are used to deal with the rest chromosomes. Since elitism preserves the fittest solution, it can speed the convergence of GA.

4.2.6 Crossover Operators

Crossover is a process of recombining chromosomes to create new offsprings, which has a great impact on the performance of GAs. Crossover can be implemented in many ways including single point crossover, two point crossover, uniform crossover, and arithmetic crossover. The selection of the method for crossover relies on the way of encoding as well as the solved problem. In this section, single point crossover for binary encoding is described.

For example, single point crossover is performed on two chromosomes 0010110011010010 and 1111111100000000. First, generate a random number as crossover point denoted by |, and then exchange the corresponding bits in two chromosomes after the crossover point. The process is shown in Table 4.3.

Table 4.3: Single point crossover for binary encoding

Chromosome 1	0010 | 110011010010
Chromosome 2	1111 | 111100000000
Offspring 1	0010 | 111100000000
Offspring 2	1111 | 110011010010

4.2.7 Mutation Operators

Mutation is a process of randomly changing a few parts of the new offspring generated by crossover operation, which prevents the solutions of the solved problem being local optimal solutions. The way of encoding usually determines the method of mutation. For a chromosome with binary encoding, first randomly choose some bits, and then switch them to 1 or 0. Table 4.4 shows the mutation operation performed on the offsprings in Table 4.3, where the mutated bit is in bold style.

Table 4.4: Mutation for binary encoding

Offspring 1	00**1**011110000**0**000
Offspring 2	1111**1**1001101001**0**
Mutated 1	00**0**011110000**1**000
Mutated 2	1111**0**1001101001**1**

4.2.8 Parameters of GAs

The settings of some important parameters including population size, crossover probability, and mutation probability usually influence the performance of GAs.

- **Population size** refers to the number of chromosomes in population. Generally, a too big population size will lower the efficiency of GAs; on the other hand, a too small population size will reduce the search space of GAs. It has been shown that increasing population size does not improve the performance of GAs after some limit which relies on the solved problem (Mitchell, 1998).
- **Crossover probability** refers to how many chromosomes will be selected to perform crossover. A crossover probability of 0 means all chromosomes in the new population are directly copied from the old population. A crossover probability of 1 means all chromosomes are generated by crossover. Generally, a high crossover probability is recommended.
- **Mutation probability** refers to how many bits in a chromosome will be mutated. A mutation probability of 0 means chromosomes are not changed at all. A mutation probability of 1 means each bit in a chromosome is changed, that is, GAs are equivalent to random search in that case. Therfore, a low mutation probability is usually recommended.

4.3 G-ANMI

G-ANMI uses a basic GA to search for an ANMI based optimal partition. In this section, the G-ANMI algorithm and some details of its implementation, including the encoding method, fitness function, selection operator, crossover operator, mutation operator, and parameters setting are described.

4.3.1 The Description of G-ANMI Algorithm

Initially, the algorithm randomly generates a set of partitions of objects and encodes them as chromosomes. These chromosomes form the initial population. Then, ANMI is used to calculate the fitness of each chromosome. Based on the fitness values, some genetic operations are employed to generate a new population by changing the chromosomes in the current population. The latest chromosomes are expected to be more similar to the optimal partition than the previous chromosomes. Repeat the above steps until the best fitness has kept invariable in some successive iterations. Figure 4.4 shows the details of G-ANMI algorithm, where M is the population size, m and r are the the mutation and crossover probability, respectively.

```
initialize the population of genetic algorithm
while (true)
    compute the fitness of chromosomes in the population;
    if (there has been no relative improvement in best fitness value for
        $N_{\max}$ iterations)
    then
        output the partition of $K_{\text{best}}$;
        exit;
    copy fittest $(1 - r - m)M$ chromosomes to new population;
    select probabilistically $\max\{2, rM\}$ chromosomes to cross over;
    apply crossover operator to the selected chromosomes
        and copy the offspring to the new population;
    select with uniform probability $\max\{1, mM\}$ chromosomes to mutate;
    apply mutation operator to the selected chromosomes
        and copy the modified chromosomes to the new population;
    Use the new population  to replace the old one;
```

Figure 4.4: The G-ANMI algorithm

4.3.2 The Encoding Method and Initialization

G-ANMI searches for the optimal partition of objects, thus each chromosome is encoded as a partition of objects. If the desired number of clusters is set to *k*, then each chromosome is encoded as a *k*-partition of objects. In the G-ANMI algorithm, the integers from interval [0, *k*-1] are used as class identifier; therefore, a chromosome is a string of integers which are in interval [0, *k*-1]. For example, suppose the number of objects is 20, and *k* is 4, a possible chromosome is as follows

1 0 2 0 1 0 3 2 3 1 0 1 2 0 3 2 0 1 1 2

The initial population consists of a set of randomly generated chromosomes. Each location of a chromosome is filled by a random number in interval [0, *k*-1].

4.3.3 The Fitness Function

G-ANMI aims to discover the optimal partition which shares most information with the partitions defined by attributes. It borrows the concept of ANMI from the algorithms *ccdByEnsemble* and *k*-ANMI to measure how well a partition summarizes the attribute partitions. Given a set of *m* partitions defined by attributes: $\Psi = \{\lambda^{(q)} \mid q \in \{1,2,...,m\}\}$ and a partition $\overline{\lambda}$, the average normalized mutual information (ANMI) between Ψ and $\overline{\lambda}$ is defined as follows:

$$\phi^{(ANMI)}(\Psi, \overline{\lambda}) = \frac{1}{m}\sum_{q=1}^{m} \phi^{(NMI)}(\overline{\lambda}, \lambda^{(q)}) \tag{4.1}$$

where $\phi^{(NMI)}(\overline{\lambda}, \lambda^{(q)})$ denotes the Normalized Mutual Information (NMI) between $\lambda^{(q)}$ and $\overline{\lambda}$. In general, given any two partitions $\lambda^{(a)}$ and $\lambda^{(b)}$, the normalized mutual information between them is calculated as follows:

$$\phi^{(NMI)}(\lambda^{(a)}, \lambda^{(b)}) = \frac{2}{n}\sum_{h=1}^{k^{(a)}}\sum_{g=1}^{k^{(b)}} n_g^{(h)} \log_{k^{(a)} * k^{(b)}}\left(\frac{n_g^{(h)} n}{n^{(h)} n_g}\right) \tag{4.2}$$

where $k^{(a)}$ and $k^{(b)}$ denote the number of clusters in partitions $\lambda^{(a)}$ and $\lambda^{(b)}$, respectively. n_g is the length of cluster C_g in $\lambda^{(b)}$, $n^{(h)}$ is the length of cluster C_h in $\lambda^{(a)}$, $n_g^{(h)}$ is the number of common objects between cluster C_g and C_h.

ANMI described in Eq. (4.1) is used to evaluate the fitness of each chromosome in population (the chromosome is viewed as $\overline{\lambda}$). The higher the ANMI of a chromosome is, the more suitable the chromosome is.

4.3.4 Selection, Crossover, and Mutation Operators

G-ANMI uses elitism method to do selection. The fittest (1-*r*-*m*)*M* chromosomes are copied directly to the new population, ensuring that current population always has the best chromosomes from the old population. Next, a number of max{2, *rM*} chromosomes from the old generation are selected probabilistically to be used in the generation of the new offspring by crossover. The selection method used is roulette wheel strategy introduced in Section 4.2.5.

The classical single point crossover operator is used in G-ANMI, which is similar to the one introduced in Section 4.2.6 for binary encoding. Starting from two chromosomes, a random crossing point is selected as a number *l* between 1 and *N* (*N* is the length of a chromosome). The offspring will contain the first 1 to *l* positions from the first parent and the last *l*+1 to N positions from the second parent and vice versa.

G-ANMI uses the classical mutation operator which involves changing randomly a number of max{1, 0.1*N*} positions in the chromosomes. The new value for each chromosome position is chosen randomly from 0 to *k*-1.

4.4 IG-ANMI

IG-ANMI improves G-ANMI by developing a new initialization method. The initial set of chromosomes are no longer entirely generated randomly, part of them are generated from the partitions defined by attributes instead. The genetic algorithm used in IG-ANMI is the same as that in G-ANMI, hence this section mainly describes the new initialization algorithm of IG-ANMI and gives an illustrative example.

The basic idea of the initialization algorithm is that integrating some equivalence classes of the partitions defined by attributes into the generation of initial partitions. Two cases are considered: one is the population size is greater than or equal to the number of attributes, another is the population size is less than the number of attributes. Different strategies are used in the two cases. The basic initialization algorithm of IG-ANMI is described in Figure 4.5. The algorithm takes *M* partitions corresponding to *M* attributes, the number of equivalence classes in each partition, and the size of population as the input and will terminate when *P* chromosomes are obtained. Each partition partitions[*i*], *i* = 0, …, *M* consists of num_eqc[*i*] equivalence classes which are labeled by 0, 1, …, num_eqc[*i*]-1 in turn. Suppose the number of clusters is set to *K*, the algorithm outputs *P* *K*-partitions each of which is a string of integer in the interval [0, *K*-1].

```
Algorithm: Initialization
Input:  partitions[i], i=0, …, M       //M partitions corresponding to M attributes,
        num_eqc[i], i=0, …, M          //The number of equivalence classes in
                                       //partitions[i],
        P  //The size of population.
Output: chrom[i], i=0, …, P         // P chromosomes (K-partitions)
Begin
        if  P >= M
            //Generate first M chromosomes from the input M partitions
              Generate_from_Partitions(M);
            //Randomly generate P-M chromosomes
              Randomly_Generate(P-M);
        else
            //Generate P chromosomes from the first P partitions
              Generate_from_Partitions(P);
        Output P chromosomes.
End.
```

Figure 4.5: The initialization algorithm of IG_ANMI

Generating chromosomes from the input partitions is implemented by a one-one way, namely one chromosome is generated by one partition. Generating a chromosome from a partition means taking some equivalence classes in the partition as the part of the chromosome. How many equivalence classes should be taken depends on the number of equivalence classes in the partition and the specified number of clusters. Different strategies are employed when the number of equivalence classes in the partition is greater than, less than, equals to the specified number of clusters, respectively. The details of function *Generate_from_Partitions* are described in Figure 4.6.

```
Function Generate_from_Partitions (int Num)
Input:    Num   //The number of chromosomes need to be generated
Output: chrom[i], i=0, …, Num    //Num chromosomes
Begin
        For each of first Num partitions, partitions[i]     // i=0, …, Num
                if num_eqc[i] equals K
                        Copy partitions[i] to chrom[i]
                else
                        if num_eqc[i] is greater than K
                                Copy first K equivalence classes of partition[i] to the
                                    corresponding location in chrom[i].
                                Generate a random number in [0, K-1] for each of the remaining
                                    locations in chrom[i].
                        else
                                Find a highest H which satisfies the following inequation
                                        N - Sum >= K-H-1
                                //where N denotes the length of a chromosome,
                                //Sum is the summation of the size of first H+1 equivalence
                                //classes of partition[i].
                                if such H can not be found
                                        // randomly generate a K-partition for chrom[i]
                                        Randomly_Generate(1)
                                else
                                        Copy first H+1 equivalence classes of partition[i] to the
                                            corresponding locations in chrom[i].
                                        Generate a random number in [H+1, K-1] for each of the
                                            remaining locations in chrom[i].
End.
```

Figure 4.6: The function of *Generate_from_Partitions*

There are two remarks on the process when num_eqc[*i*] is less than *K:*

(1) When num_eqc[*i*] is less than *K*, the original idea is to copy first num_eqc[*i*]-1 equivalence classes to chrom[*i*] at first, and then generate a random number in interval [num_eqc[*i*]-1, *K*-1] for each of the remaining locations in chrom[*i*]. However, this method can not make sure if the remaining locations are enough for each number in interval [num_eqc[*i*]-1, *K*-1] appears at least once in chrom[*i*], namely it can not make sure chrom[*i*] is a *K*-partition. Thus an appropriate number *H* have to be found so that the remaining locations are enough for each number in interval [*H*+1, *K*-1] appears at least once in chrom[*i*] after copying first *H*+1 (labeled by 0, 1, …, *H*) equivalence classes to chrom[*i*]. Such *H* satisfies the inequation *N*-Sum >= *K*-*H*-1, where *N*-Sum denotes the number of remaining locations in chrom[*i*] after copying first *H*+1 equivalence classes to chrom[*i*], *K*-*H*-1 is the length of interval [*H*+1, *K*-1]. In order to copy as many as possible equivalence classes to chrom[*i*], the highest *H* is selected.

It is possible that any *H* which satisfies the above inequation can not be found. That means the remaining locations are not enough for each number in interval [1, *K*-1] appears at least once in chrom[*i*] after copying the first (labeled by 0) equivalence class to chrom[*i*]. There are many ways to deal with such case. In our implementation, a *K*-partition is randomly generated for chrom[*i*].

(2) Even if an appropriate number *H* is found, the algorithm has to make sure each number in interval [*H*+1, *K*-1] appears at least once in chrom[*i*] when generating a random number for each of the remaining locations in chrom[*i*].

When the population size *P* is greater than the number of attributes *M*, the latter *P*-*M* chromosomes are generated randomly. Each chromosome is a string of integers which are in interval [0, *K*-1]. The details of function *Randomly_Generate* are described in Figure 4.7. Note that the algorithm must make sure all the numbers in interval [0, *K*-1] appear at least once in a chromosome, or else the chromosome is not a *K*-partition.

```
Function Randomly_Generate(int Num)
Input:   Num  //The number of chromosomes need to be generated
Output: chrom[i], i=0, …, Num   //Num chromosomes
Begin
    i=0
    while i<Num
        for each location of the chrom[i]
            Generate a random number in [0, K-1]
        if all the numbers in [0, K-1] appear in the chrom[i]
            i++
End.
```

Figure 4.7: The function of *Randomly_Generate*

An illustrative example of the IG-ANMI algorithm is as follows.

Example 4.1. Suppose a data set has ten objects with four attributes, namely N=10, M=4. Table 4.5 shows the partitions defined by the four attributes. The numbers 0, 1, 2, and 3 in the partitions denote different equivalence classes (categories). The algorithm IG-ANMI is used to cluster the objects below. The parameter setting includes: the number of clusters K=3, the population size P=10, random seed=1, mutation probability=0.1, crossover probability=0.8 and $N_{\max}$=100.

Table 4.5: The example of partitions defined by four attributes

U	A_0	A_1	A_2	A_3
O_1	0	0	0	0
O_2	1	0	1	1
O_3	0	1	0	0
O_4	0	0	0	0
O_5	1	1	2	2
O_6	1	1	1	2
O_7	2	0	2	3
O_8	2	1	2	1
O_9	1	1	1	2
O_{10}	2	1	2	3

Following the initialization algorithm shown in Figure 4.5, since the population size *P* is greater than the number of attributes *M*, first four chromosomes are generated by using the four partitions associated with the attributes and remaining six chromosomes randomly. The partitions defined by the four attributes have 3, 2, 3, and 4 equivalence classes, respectively. The numbers of equivalence classes in attributes A_0 and A_2 equal the specified number of clusters *K*, so the partitions of attributes A_0 and A_2 are directly copied to chrom [0] and chrom [2], respectively. For the attribute A_1, the number of equivalence classes in its partitions is less than *K*. According to the function shown in Figure 4.6, an appropriate number *H* should be found first. In this example, there is only possible value for *H*, namely zero. Zero satisfies N - Sum >= K-H-1, thus *H* get the value zero. Next, the first equivalence class is copied to the corresponding location in chrom [1]. Table 4.6 shows the status of chrom [1] after copying the first equivalence class.

Table 4.6: The status of chrom [1] after copying the first equivalence class

location	0	1	2	3	4	5	6	7	8	9
chrom[1]	0	0		0			0			

There are still six locations need to be filled in chrom [1]. A random number in interval [1, 2] is generated for each of the six locations.

For the attribute A_3, the number of equivalence classes in its partitions is greater than *K*. According to the function *Generate_from_Partitions* shown in Figure 4.6, first three equivalence classes of partitions [3] are copied to the corresponding location in chrom [3]. Table 4.7 shows the status of chrom [3] after copying first three equivalence classes.

Table 4.7: The status of chrom [3] after copying first three equivalence classes

location	0	1	2	3	4	5	6	7	8	9
chrom[3]	0	1	0	0	2	2		1	2	

There are still two locations need to be filled in chrom [3]. A random number in interval [0, 2] is generated for each of the two locations. The first four chromosomes are obtained at the end of the function of *Generate_from_Partitions* and summarized in Table 4.8. The numbers in bold style are randomly generated.

Table 4.8: The first four chromosomes generated from attributes partitions

location	**chrom[0]**	**chrom[1]**	**chrom[2]**	**chrom[3]**
0	0	0	0	0
1	1	0	1	1
2	0	**1**	0	0
3	0	0	0	0
4	1	**1**	2	2
5	1	**1**	1	2
6	2	0	2	**0**
7	2	**1**	2	1
8	1	**1**	1	2
9	2	**2**	2	**1**

Following the initialization algorithm, the remaining six chromosomes will be randomly generated. According to the function *Randomly_Generate*, six chromosomes are obtained and summarized in Table 4.9.

Table 4.9: The six chromosomes generated randomly

location	**chrom[4]**	**chrom[5]**	**chrom[6]**	**chrom[7]**	**chrom[8]**	**chrom[9]**
0	1	2	1	2	1	1
1	1	0	1	1	1	2
2	1	2	0	0	0	0
3	2	1	1	1	1	0
4	1	2	1	1	0	2
5	1	2	1	1	0	0
6	0	2	2	0	1	0
7	1	2	2	0	0	1
8	0	0	1	2	2	2
9	0	0	2	1	0	0

Note that the equivalence classes in each of the first four chromosomes are labeled by order 0, 1, 2. However, the equivalence classes in each of other six chromosomes are labeled unorderly. Actually, the numbers 0, 1, 2 in the partions or chromosomes only denote different categories rather than order. That means the order of the labels doesn't affect the computation of fitness of a chromosome. Even if the order of the labels is changed in some chromosomes, their fitness values keep invarible. For instance, chrom[0] can be changed from {0, 1, 0, 0, 1, 1, 2, 2, 1, 2} to {1, 2, 1, 1, 2, 2, 0, 0, 2, 0}, chrom[4] can be changed from {1, 1, 1, 2, 1, 1, 0, 1, 0, 0} to {0, 0, 0, 1, 0, 0, 2, 0, 2, 2}, and so on.

After the initialization, the next step of IG-ANMI is to compute the fitness for chromosomes. Let us take chrom [0] as an example. First, the NMI between chrom [0] and the attributes are calculated by using Eq. (4.2). The calculation of NMI between chrom [0] and attribute A_0 is as follows.

Let the partition defined by chrom [0] be $\lambda^{(a)}$, the partition defined by A_0 be $\lambda^{(b)}$. As mentioned above, chrom [0] is a copy of the partition defined by A_0, that is, $\lambda^{(a)} = \lambda^{(b)} = \{\{O_1, O_3, O_4\}, \{O_2, O_5, O_6, O_9\}, \{O_7, O_8, O_{10}\}\}$. Therefore, $k^{(a)} = k^{(b)} = 3$, $n^{(1)} = 3$, $n^{(2)} = 4$, $n^{(3)} = 3$, $n_1 = 3$, $n_2 = 4$, $n_3 = 3$, $n_1^{(1)} = 3$, $n_2^{(1)} = 0$, $n_3^{(1)} = 0$, $n_1^{(2)} = 0$, $n_2^{(2)} = 4$, $n_3^{(2)} = 0$, $n_1^{(3)} = 0$, $n_2^{(3)} = 0$, $n_3^{(3)} = 3$, and $n = 10$. Take these values into Eq. (4.2).

$$\phi^{(NMI)}(\lambda^{(a)},\lambda^{(b)}) = \frac{2}{10} \times (3\log_9 \frac{3\times 10}{3\times 3} + 4\log_9 \frac{4\times 10}{4\times 4} + 3\log_9 \frac{3\times 10}{3\times 3}) = 0.991159$$

That is, the NMI between chrom [0] and attribute A_0 is 0.991159. With the same process, the NMI between chrom [0] and other three attribute are obtained. The NMI between chrom [0] and A_1, A_2, A_3 are 0.073859, 0.786416 and 0.764834, respectively. Next, using the Eq. (4.1), the ANMI between chrom [0] and attribute partitions is calculated as follows

$$\frac{1}{4}(0.991159 + 0.073859 + 0.786416 + 0.764834) = 0.654067$$

That is, the fitness of chrom [0] is 0.654067. Repeating the above process on other chromosomes, their fitnesses are obtained as is shown in Table 4.10.

Table 4.10: The fitness of initial chromosomes of IG-ANMI

fitness[i]	**fitness value**	**average**
fitness[0]	0.654067	0.53927
fitness[1]	0.361562	
fitness[2]	0.615644	
fitness[3]	0.525807	
fitness[4]	0.252361	0.265807
fitness[5]	0.196573	
fitness[6]	0.403407	
fitness[7]	0.166014	
fitness[8]	0.323139	
fitness[9]	0.253349	
average	0.375192	

It can be seen from Table 4.10 that the first four chromosomes has higher average fitness value compared with other six chromosomes, which indicates that the chromosomes generated from the partitions associated with attributes are closer to the optimal partition than that generated randomly. With these fitness values, the algorithm IG-ANMI continues to generate new population, namely modify the chromosomes in the current population by using crossover and mutation as genetic operators. Table 4.11

shows the first new population after initial population. Chrom [0] has the highest fitness in the initial population. From the Table 4.11, it can be seen that several chromosomes which are dissimilar to chrom [0] in the initial population are very close to chrom [0] in the new population, such as chrom [5], chrom [6], and chrom [8]. Table 4.12 shows the fitness values of the chromosomes in the new population. The average fitness of the new chromosomes is obviously higher than that of initial chromosomes. After five iterations, all the chromosomes are the same, namely {0, 1, 0, 0, 1, 1, 2, 2, 1, 2} and have the same fitness 0.654067. Since the best fitness keep invariable during 100 successive iterations, the algorithm IG-ANMI ends after the 100th iteration. Finally, IG-ANMI produces the optimal 3-partition {0, 1, 0, 0, 1, 1, 2, 2, 1, 2}.

Table 4.11: The first new population after initial population in IG-ANMI

location	chrom[0]	chrom[1]	chrom[2]	chrom[3]	chrom[4]	chrom[5]	chrom[6]	chrom[7]	chrom[8]	chrom[9]
0	0	1	0	0	1	0	0	1	0	0
1	1	1	1	1	2	1	1	1	1	0
2	0	0	0	0	0	0	0	0	0	1
3	0	0	1	0	0	0	0	1	0	0
4	1	1	0	2	2	1	2	1	2	1
5	1	1	0	0	1	1	1	2	1	1
6	2	2	1	0	2	2	2	0	2	0
7	2	2	0	1	2	2	2	1	2	1
8	1	1	2	2	1	1	1	2	1	1
9	2	2	0	0	2	2	2	1	2	2

Table 4.12: The fitness values of the chromosomes in the first new population

fitness[i]	**fitness value**
fitness[0]	0.654067
fitness[1]	0.473318
fitness[2]	0.217901
fitness[3]	0.310606
fitness[4]	0.407429
fitness[5]	0.654067
fitness[6]	0.615644
fitness[7]	0.263929
fitness[8]	0.615644
fitness[9]	0.361562
average	0.457417

G-ANMI is used to cluster the same data set below. The parameter setting of G-ANMI is as the same as the parameter setting of IG-ANMI. Firstly, G-ANMI randomly generates *P* chromosomes as shown in Table 4.13. Table 4.14 shows the fitness values of the chromosomes in the initial population. Obviously, the average fitness as well as the best fitness of the chromosomes is less than that in the initial population generated by algorithm IG-ANMI. After 27 iterations, the best fitness reaches 0.654067, which equals the best fitness of the initial population generated by algorithm IG-ANMI. Since the best fitness keep invariable during the subsequent 99 successive iterations, the algorithm G-ANMI ends after the 127th iterations. G-ANMI requires 27 more iterations than IG-ANMI due to the randomly generated initial population.

Table 4.13: The initial population of G-ANMI

location	chrom[0]	chrom[1]	chrom[2]	chrom[3]	chrom[4]	chrom[5]	chrom[6]	chrom[7]	chrom[8]	chrom[9]
0	1	1	2	0	0	0	0	0	2	2
1	1	2	1	1	2	1	1	0	1	1
2	2	1	2	1	2	1	0	2	1	0
3	1	1	2	1	0	1	0	0	2	1
4	1	0	2	2	0	0	1	0	2	2
5	1	1	2	2	0	0	0	1	1	1
6	0	0	0	1	2	2	2	2	2	2
7	1	0	0	2	0	1	0	0	0	1
8	1	2	1	0	2	1	2	1	1	2
9	1	0	1	0	1	1	1	1	0	0

Table 4.14: The fitness values of the chromosomes in the initial population of G-ANMI

fitness[i]	**fitness value**
fitness[0]	0.211672
fitness[1]	0.469059
fitness[2]	0.338877
fitness[3]	0.227614
fitness[4]	0.139958
fitness[5]	0.181013
fitness[6]	0.216344
fitness[7]	0.263182
fitness[8]	0.377376
fitness[9]	0.166627
average	0.259172

4.5 SUMMARY

This chapter introduces the basic genetic algorithm, analyzes the reason for the low efficiency of G-ANMI algorithm and proposes an improved genetic clustering algorithm named IG-ANMI for categorical data. An attribute-oriented initialization method is used in IG-ANMI, in which attributes partitions are integrated into the generation of initial chromosomes. The example illustrates that IG-ANMI converges faster than G-ANMI, which implies that IG-ANMI has higher efficiency than G-ANMI.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 INTRODUCTION

A set of experiments are performed to evaluate clustering performance, clustering efficiency and scalability of MGR and IG-ANMI algorithms. In this chapter, the details of these experiments and the corresponding results are presented, and MGR and IG-ANMI algorithms are compared with MMR, k-ANMI, G-ANMI, and COOLCAT algorithms in terms of clustering performance and efficiency. The chapter is structured as follows:

- Section 5.2 introduces the experiments design, including algorithms used for comparison, data sets, parameters setting, and the platform for conducting experiments.
- Section 5.3 describes the performance analysis and comparison of six algorithms on four real-life data sets.
- Section 5.4 describes the efficiency analysis and comparison of six algorithms on ten synthetic data sets.
- Section 5.5 describes the running results of MGR algorithm without specifying the number of clusters.
- Section 5.6 describes the scalability of MGR and IG-ANMI algorithms.
- Section 5.7 describes the comparison between IG-ANMI and G-ANMI.

5.2 EXPERIMENT DESIGN

5.2.1 Algorithms Used for Comparison

Besides our algorithms MGR and IG-ANMI, other four algorithms including MMR, k-ANMI, G-ANMI, and COOLCAT are repeated for comparing MGR and IG-ANMI with them. Choosing these algorithms for comparison is based on the following consideration:

- MMR is the most similar work to our MGR algorithm among all existing algorithms for categorical data clustering.
- IG-ANMI improves G-ANMI by proposing a new initialization method.
- COOLCAT, k-ANMI, and G-ANMI, are also based on information theory.
- k-ANMI and G-ANMI also explore the relation between the partitions although the scopes of partitions they use are much larger than that MGR uses.

5.2.2 Data Sets Used in Experiments

i. Real life data sets

Four real-life data sets derived from the UCI Machine Learning Repository (http://www.ics.uci.edu_/mlearn/MLRepository.html) are employed to appraise the clustering performance, including the Zoo data set, the Congressional Votes data set (Votes for short), the Wisconsin Breast Cancer data set (Breast Cancer for short), and the Mushroom data set. The reason for choosing these four datasets is that they are also partially or completely used in MMR, k-ANMI, G-ANMI, and COOLCAT algorithms for evaluation. A brief introduction about these data sets is given as follows.

- **Zoo data set**

There are 101 instances of animals in the Zoo data set. Each animal is characterized by 17 attributes, where the first attribute denotes the name of animals (does not participate in the clustering), 15 Boolean attributes denote the presence of some characters such as fins, feathers and so on, 1 non-Boolean attribute represents the number of legs. These animals can be categorized into 7 classes, that is, fish, mammal, bird, invertebrate, insect, reptile, and amphibian.

- **Congressional Votes data set**

The data set contains the Congressional Voting Records of United States in 1984. Each record (a vote) is characterized by 16 Boolean attributes. Each record is classified into two classes: Republican and Democrat. There are 435 records in this data set including 267 Democrats and 168 Republicans.

- **Wisconsin Breast Cancer data set**

The data set derived from the University Medical Center, Institute of Oncology, Ljubljana, Yugoslavia. There are 699 instances in this data set, each of which is described by 9 attributes. Each instance is classified into two classes: Benign (458 instances) and Malignant (241 instances).

- **Mushroom data set**

There are 8124 instances in this data set, each of which represents a mushroom. The physical characters of each mushroom are described by 22 categorical attributes. Each mushroom is classified into two classes: poisonous (3916 instances) and edible (4208 instances).

The basic information about these data sets is summarized in Table 5.1.

Table 5.1: The basic information about the four data sets

Data set	**Number of objects**	**Number of Attributes**	**Number of classes**
Zoo	101	16	7
Votes	435	16	2
Breast Cancer	699	9	2
Mushroom	8124	22	2

Except for the Zoo data set, there are missing values in other three data sets. Since algorithms k-ANMI and G-ANMI delete the objects with missing value in the

Breast Cancer data, the same process is performed on the Breast Cancer data set in our experiment for a fair comparison between them and other algorithms. Thus, the Breast Cancer data set used in our experiments contains 683 objects with 444 Benign and 239 Malignant. For the Votes and Mushroom data sets, missing values are considered as particular categories.

ii. Synthetically generated data sets

Using the method proposed by Cristofor and Simovici (2002) for synthetically generating dataset, 10 categorical datasets are created to evaluate the efficiency and test the scalability of MGR and IG-ANMI algorithms. The generation of these data sets follows the pattern: for each object number *objectid* $\in$ [0, N-1], a number $i \in$ [0, K-1] is randomly generated and saved at position *objectid* in the *reference* partition (*reference* partition is regarded as real clusters of the objects) and in all attribute partitions, but one. The exception attribute $a \in A$, randomly chosen, receives at position *objectid* a different value $j \in$ [0, K-1], $j \neq i$. To ensure that the *reference* and attribute partitions have exactly K classes, the first values for i are 0, 1, …, K-1. These ten datasets contain 10,000, 20,000 through 100,000 instances, respectively. The number of attributes and classes are set to be 10 and 10 separately. These datasets are named by *R*1, *R*2 through *R*10.

5.2.3 Parameters Setting

Six algorithms are sequentially run on all the data sets. Each algorithm has some parameters which need to be set before running.

MMR, k-ANMI, G-ANMI, IG-ANMI, and COOLCAT have a common input parameter, namely the number of clusters. It is set to be as the same as the number of classes provided with the data set in our experiments. For example there are 7 classes in the Zoo data set, so the number of clusters in these algorithms is set to 7. As mentioned in Chapter 3, MGR algorithm can be run with specifying the number of clusters as well as without specifying the number of clusters. For the comparison purpose, MGR is assigned as the same number of clusters as that for other five algorithms (Section 5.5 describes the experimental results of MGR without specifying the number of clusters).

In all the experiments, the *threshold* of the size of the splitting equivalence class in MGR algorithm is set to be 3% of the number of objects in data set.

All the parameters required by G-ANMI and IG-ANMI are set to be default as in G-ANMI. Concretely, the parameter setteing includes: random seed=1, mutation probability=0.1, crossover probability=0.8 and $N_{\max}$=100. Moreover, population size greatly affects the quality of clustering in these two algorithms. Therefore, the population size is varied from 50 to 500 on each data set (from 50 to 200 on Mushroom data set because G-ANMI and IG-ANMI are very time-consuming on Mushroom data set when the population size is greater than 200) to calculate the average performance and efficiency for comparison to other algorithms as well as comparison between IG-ANMI and G-ANMI.

Besides the number of clusters, COOLCAT has two other parameters: buffer size and the percent of reprocess. Since each dataset has different number of objects, different buffer size is specified for each dataset. The buffer size is set to be 30, 100, 100, and 200 for Zoo, Votes, Breast Cancer, and Mushroom data sets, respectively. The buffer size is set to be 2% of the number of objects for each synthetically generated data set. The percent of reprocess greatly affects the quality of clustering in COOLCAT. In our implementation, the percent of reprocess is set to 0, 10%, 20%, and 40% respectively to calculate the average performance and efficiency.

5.2.4 Language and Platform for Implementation

All the algorithms are coded in C language and compiled on the Borland C++ version 5.02. All experiments are conducted on a machine with Intel Core2 Duo CPU T7250 @ 2.00GHz, 1.99 GB of RAM, running Microsoft Windows XP Professional.

5.3 PERFORMANCE ANALYSIS

5.3.1 Evaluation Method

It is an important task to validate clustering results (He et al., 2008). One of the commonly used ways to appraise the results of clustering algorithms is clustering accuracy (Huang, 1998), also named clustering purity. This method needs external class labels to compute the best matches between the clusters produced by clustering algorithms and the true clusters. Given the true class labels, clustering accuracy is formally defined as follows:

$$\text{Clustering accuracy} = \frac{\sum_{i=1}^{k} a_i}{n} \tag{5.1}$$

where k denotes the specified number of clusters, a_i is size of the class having the most objects in the i^{th} cluster, n denotes the number of instances in the used data set. According to this measure, a clustering accuracy of 1 means the objects in each cluster have the same class label, which is an expected result. Hence, the higher the clustering accuracy is, the better the clustering result (He et al., 2008).

Note that the clustering error defined as $1 - \frac{\sum_{i=1}^{k} a_i}{n}$ is also used in many literatures. It is essentially the same as clustering accuracy. In addition, some famous clustering evaluation methods are described in the literature (Mali and Mitra, 2003). As pointed out by He et al. (2008), these methods coincide with clustering accuracy (error). Thus, only clustering accuracy is used in our experiments to appraise the algorithms' performance.

Based on the evaluation measure of clustering accuracy, the performance of algorithms MGR and IG-ANMI and their comparison with other algorithms on four real life data sets are presented as follows.

5.3.2 Zoo Data Set

The results of MGR, MMR and k-ANMI algorithms on Zoo data set are shown in Tables 5.2-5.4, respectively.

Table 5.2: The results of MGR algorithm on the Zoo data set

No. of Cluster	Mammal	Fish	Bird	Invertebrate	Insect	Amphibian	Reptile	Max Number	Accuracy
1	41	0	0	0	0	0	0	41	
2	0	13	0	0	0	0	0	13	
3	0	0	20	0	0	0	0	20	
4	0	0	0	0	4	0	0	4	0.931
5	0	0	0	0	0	4	5	5	
6	0	0	0	7	0	0	0	7	
7	0	0	0	3	4	0	0	4	

Table 5.3: The results of MMR algorithm on the Zoo data set

No. of Cluster	Mammal	Fish	Bird	Invertebrate	Insect	Amphibian	Reptile	Max Number	Accuracy
1	39	0	0	0	0	0	0	39	
2	0	0	20	0	0	0	0	20	
3	2	0	0	0	6	0	0	6	
4	0	0	0	10	2	0	0	10	0.911
5	0	13	0	0	0	0	1	13	
6	0	0	0	0	0	1	1	1	
7	0	0	0	0	0	3	3	3	

Table 5.4: The results of k-ANMI algorithm on the Zoo data set

No. of Cluster	Mammal	Fish	Bird	Invertebrate	Insect	Amphibian	Reptile	Max Number	Accuracy
1	5	0	0	0	0	0	0	5	
2	6	0	0	0	0	0	0	6	
3	0	13	0	8	0	4	4	13	
4	15	0	0	0	0	0	0	15	0.733
5	4	0	0	0	0	0	0	4	
6	11	0	0	0	0	0	0	11	
7	0	0	20	2	8	0	1	20	

The clustering accuracy of MGR is 0.931, which means 93.1% of animals have the dominant class label of the cluster in which they are grouped. The clustering accuracy of MMR and k-ANMI are 0.911 and 0.733, respectively.

The clustering accuracy of G-ANMI and IG-ANMI vary with different population sizes. Tables 5.5 and 5.6 show the clustering results of G-ANMI and IG-ANMI respectively when the population size is set to 50. Table 5.7 and Table 5.8 summarize the clustering accuracy of G-ANMI and IG-ANMI respectively as the population size is increased from 50 with step 50 up to 500.

Table 5.5: The results of G-ANMI algorithm on the Zoo data set when the population size is set to 50

No. of Cluster	Mammal	Fish	Bird	Invertebrate	Insect	Amphibian	Reptile	Max Number	Accuracy
1	18	0	0	0	0	0	0	18	
2	18	0	0	0	0	0	0	18	
3	0	13	0	0	0	0	1	13	
4	0	0	4	4	8	0	1	8	0.832
5	0	0	0	6	0	2	3	6	
6	0	0	16	0	0	0	0	16	
7	5	0	0	0	0	2	0	5	

Table 5.6: The results of IG-ANMI algorithm on the Zoo data set when the population size is set to 50

No. of Cluster	Mammal	Fish	Bird	Invertebrate	Insect	Amphibian	Reptile	Max Number	Accuracy
1	12	0	0	0	0	0	0	12	
2	26	0	0	0	0	0	0	26	
3	3	13	0	0	0	1	4	13	
4	0	0	20	0	0	0	0	20	0.881
5	0	0	0	5	0	0	1	5	
6	0	0	0	5	0	3	0	5	
7	0	0	0	0	8	0	0	8	

Table 5.7: The clustering accuracy of G-ANMI on the Zoo data set with the increase of the population size

Population size	50	100	150	200	250	300	350	400	450	500	Average
Accuracy	0.832	0.851	0.911	0.881	0.861	0.891	0.931	0.812	0.851	0.921	0.874

Table 5.8: The clustering accuracy of IG-ANMI on the Zoo data set with the increase of the population size

Population size	50	100	150	200	250	300	350	400	450	500	Average
Accuracy	0.881	0.911	0.901	0.911	0.921	0.921	0.921	0.931	0.931	0.931	0.916

The clustering accuracy of COOLCAT varies with different percent of reprocess. Table 5.9 shows the clustering results of COOLCAT when percent of reprocess is set to 10%. Table 5.10 summarizes the clustering accuracy of COOLCAT when the percent of reprocess are 0, 10%, 20%, and 40%, respectively.

Table 5.9: The results of COOLCAT algorithm on the Zoo data set when the percent of reprocess is set to 10%

No. of Cluster	**Mammal**	**Fish**	**Bird**	**Invertebrate**	**Insect**	**Amphibian**	**Reptile**	**Max Number**	**Accuracy**
1	11	0	1	0	1	0	0	11	
2	0	0	0	5	2	1	0	5	
3	0	0	18	1	1	0	2	18	
4	1	13	1	0	1	1	2	13	0.792
5	17	0	0	0	1	0	1	17	
6	0	0	0	4	1	1	0	4	
7	12	0	0	0	1	1	0	12	

Table 5.10: The clustering accuracy of COOLCAT on the Zoo data set when the percent of reprocess are 0, 10%, 20%, and 40%

Percent of reprocess	0	10%	20%	40%	Average
Accuracy	0.792	0.792	0.772	0.782	0.785

5.3.3 Votes Data Set

The results of MGR, MMR and k-ANMI algorithms on Votes data set are shown in Tables 5.11-5.13, respectively.

Table 5.11: The results of MGR algorithm on the Votes data set

No. of Cluster	**Votes**	**Republicans**	**Democrats**	**Max Number**	**Accuracy**
1	208	8	200	200	0.828
2	227	160	67	160	

Table 5.12: The results of MMR algorithm on the Votes data set

No. of Cluster	**Votes**	**Republicans**	**Democrats**	**Max Number**	**Accuracy**
1	236	134	102	134	0.687
2	199	34	165	165	

Table 5.13: The results of k-ANMI algorithm on the Votes data set

No. of Cluster	**Votes**	**Republicans**	**Democrats**	**Max Number**	**Accuracy**
1	207	159	48	159	0.869
2	228	9	219	219	

The clustering accuracy of G-ANMI and IG-ANMI vary with different population sizes. Tables 5.14 and 5.15 show the clustering results of G-ANMI and IG-ANMI respectively when the population size is set to 50. Table 5.16 and Table 5.17 summarize the clustering accuracy of G-ANMI and IG-ANMI respectively as the population size is increased from 50 with step 50 up to 500.

Table 5.14: The results of G-ANMI algorithm on the Votes data set when the population size is set to 50

No. of Cluster	Votes	Republicans	Democrats	Max Number	Accuracy
1	205	159	46	159	0.874
2	230	9	221	221	

Table 5.15: The results of IG-ANMI algorithm on the Votes data set when the population size is set to 50

No. of Cluster	Votes	Republicans	Democrats	Max Number	Accuracy
1	190	144	46	144	0.839
2	245	24	221	221	

Table 5.16: The clustering accuracy of G-ANMI on the Votes data set with the increase of the population size

Population size	50	100	150	200	250	300	350	400	450	500	Average
Accuracy	0.874	0.862	0.876	0.864	0.88	0.874	0.869	0.864	0.876	0.874	0.871

Table 5.17: The clustering accuracy of IG-ANMI on the Votes data set with the increase of the population size

Population size	50	100	150	200	250	300	350	400	450	500	Average
Accuracy	0.839	0.86	0.88	0.862	0.867	0.869	0.876	0.876	0.871	0.878	0.868

The clustering accuracy of COOLCAT varies with different percent of reprocess. Table 5.18 shows the clustering results of COOLCAT when the percent of reprocess is set to 10%. Table 5.19 summarizes the clustering accuracy of COOLCAT when the percent of reprocess are 0, 10%, 20%, and 40%, respectively.

Table 5.18: The results of COOLCAT algorithm on the Votes data set when the percent of reprocess is set to 10%

No. of Cluster	**Votes**	**Republicans**	**Democrats**	**Max Number**	**Accuracy**
1	216	160	56	160	0.853
2	219	8	211	211	

Table 5.19: The clustering accuracy of COOLCAT on the Votes data set when the percent of reprocess are 0, 10%, 20%, and 40%

Percent of reprocess	0	10%	20%	40%	Average
Accuracy	0.809	0.853	0.853	0.839	0.839

5.3.4 Breast Cancer Data Set

The results of MGR, MMR and k-ANMI algorithms on Breast Cancer data set are shown in Tables 5.20-5.22, respectively.

Table 5.20: The results of MGR algorithm on the Breast Cancer data set

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	373	369	4	369	0.884
2	310	75	235	235	

Table 5.21: The results of MMR algorithm on the Breast Cancer data set

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	13	0	13	13	0.669
2	670	444	226	444	

Table 5.22: The results of k-ANMI algorithm on the Breast Cancer data set

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	433	431	2	431	0.978
2	250	13	237	237	

The clustering accuracy of G-ANMI and IG-ANMI vary with different population sizes. Tables 5.23 and 5.24 show the clustering results of G-ANMI and IG-ANMI respectively when the population size is set to 50. Table 5.25 and Table 5.26 summarize the clustering accuracy of G-ANMI and IG-ANMI respectively as the population size is increased from 50 with step 50 up to 500.

Table 5.23: The results of G-ANMI algorithm on the Breast Cancer data set when the population size is set to 50

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	404	398	6	398	0.924
2	279	46	233	233	

Table 5.24: The results of IG-ANMI algorithm on the Breast Cancer data set when the population size is set to 50

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	438	419	19	419	0.936
2	245	25	220	220	

Table 5.25: The clustering accuracy of G-ANMI on the Breast Cancer data set with the increase of the population size

Population size	50	100	150	200	250	300	350	400	450	500	Average
Accuracy	0.924	0.962	0.96	0.963	0.972	0.975	0.975	0.975	0.977	0.978	0.966

Table 5.26: The clustering accuracy of IG-ANMI on the Breast Cancer data set with the increase of the population size

Population size	50	100	150	200	250	300	350	400	450	500	Average
Accuracy	0.936	0.966	0.965	0.969	0.971	0.977	0.975	0.977	0.972	0.978	0.969

The clustering accuracy of COOLCAT varies with different percent of reprocess. Table 5.27 shows the clustering results of COOLCAT when the percent of reprocess is set to 10%. Table 5.28 summarizes the clustering accuracy of COOLCAT when the percent of reprocess are 0, 10%, 20%, and 40%, respectively.

Table 5.27: The results of COOLCAT algorithm on the Breast Cancer data set when the percent of reprocess is set to 10%

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	149	83	66	83	0.65
2	534	361	173	361	

Table 5.28: The clustering accuracy of COOLCAT on the Breast Cancer data set when the percent of reprocess are 0, 10%, 20%, and 40%

Percent of reprocess	0	10%	20%	40%	Average
Accuracy	0.65	0.65	0.65	0.65	0.65

5.3.5 Mushroom Data Set

The results of MGR, MMR and k-ANMI algorithms on Mushroom data set are shown in Table 5.29-5.31, respectively.

Table 5.29: The results of MGR algorithm on the Mushroom data set

No. of Cluster	**Instances**	**Poisonous**	**Edible**	**Max Number**	**Accuracy**
1	1296	1296	0	1296	0.677
2	6828	2620	4208	4208	

Table 5.30: The results of MMR algorithm on the Mushroom data set

No. of Cluster	**Instances**	**Poisonous**	**Edible**	**Max Number**	**Accuracy**
1	7914	3898	4016	4016	0.518
2	210	18	192	192	

Table 5.31: The results of k-ANMI algorithm on the Mushroom data set

No. of Cluster	**Instances**	**Poisonous**	**Edible**	**Max Number**	**Accuracy**
1	3184	1872	1312	1872	0.587
2	4940	2044	2896	2896	

The clustering accuracy of G-ANMI and IG-ANMI vary with different population sizes. Tables 5.32 and 5.33 show the clustering results of G-ANMI and IG-

ANMI respectively when the population size is set to 50. Table 5.34 and Table 5.35 summarize the clustering accuracy of G-ANMI and IG-ANMI respectively as the population size is increased from 50 with step 50 up to 200.

Table 5.32: The results of G-ANMI algorithm on the Mushroom data set when the population size is set to 50

No. of Cluster	Instances	Poisonous	Edible	Max Number	Accuracy
1	3780	2008	1772	2008	0.547
2	4344	1908	2436	2436	

Table 5.33: The results of IG-ANMI algorithm on the Mushroom data set when the population size is set to 50

No. of Cluster	Instances	Poisonous	Edible	Max Number	Accuracy
1	3376	624	2752	2752	0.744
2	4748	3292	1456	3292	

Table 5.34: The clustering accuracy of G-ANMI on the Mushroom data set with the increase of the population size

Population size	50	100	150	200	Average
Accuracy	0.547	0.568	0.546	0.538	0.55

Table 5.35: The clustering accuracy of IG-ANMI on the Mushroom data set with the increase of the population size

Population size	50	100	150	200	Average
Accuracy	0.744	0.902	0.847	0.901	0.849

The clustering accuracy of COOLCAT varies with different percent of reprocess. Table 5.36 shows the clustering results of COOLCAT when the percent of reprocess is set to 10%. Table 5.37 summarizes the clustering accuracy of COOLCAT when the percent of reprocess are 0, 10%, 20%, and 40%, respectively.

Table 5.36: The results of COOLCAT algorithm on the Mushroom data set when the percent of reprocess is set to 10%

No. of Cluster	Instances	Poisonous	Edible	Max Number	Accuracy
1	4253	1813	2440	2440	0.559
2	3871	2103	1768	2103	

Table 5.37: The clustering accuracy of COOLCAT on the Mushroom data set when the percent of reprocess are 0, 10%, 20%, and 40%

Percent of reprocess	0	10%	20%	40%	Average
Accuracy	0.518	0.559	0.518	0.53	0.531

5.3.6 Comparison and Discussion

The clustering accuracies of these six algorithms are summarized in Table 5.38, where G-ANMI and IG-ANMI use the average clustering accuracy of different population sizes. The last column of the table shows the average clustering accuracy of each algorithm on four data sets. Figure 5.1 illustrates their comparison.

Table 5.38: The accuracy of six algorithms on four data sets

Algorithms	Zoo	Vote	Cancer	Mushroom	Average
MMR	0.911	0.687	0.669	0.518	0.696
MGR	0.931	0.828	0.884	0.677	0.83
k-ANMI	0.733	0.869	0.978	0.587	0.792
G-ANMI	0.874	0.871	0.966	0.55	0.815
COOLCAT	0.785	0.839	0.65	0.531	0.701
IG-ANMI	0.916	0.868	0.969	0.849	0.901

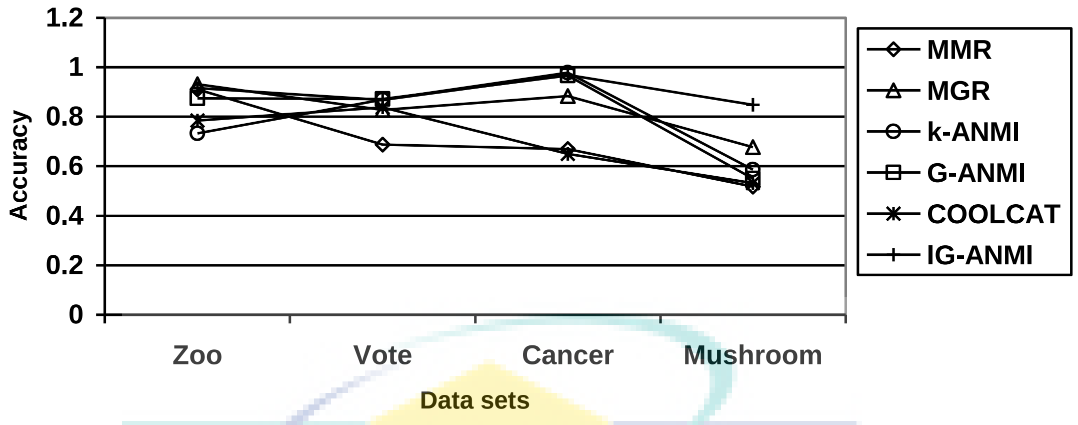


Figure 5.1: Clustering accuracy of six algorithms on four data sets

Some significant conclusions are obtained from Table 5.38 and Figure 5.1 and summarized as follows:

- IG-ANMI algorithm has the highest average clustering accuracy. MGR has the second highest average clustering accuracy.
- IG-ANMI algorithm outperforms MMR and COOLCAT on all of four data sets, outperforms G-ANMI and MGR on three data sets, and outperforms k-ANMI on two data sets. Although k-ANMI performs better than IG-ANMI on Votes and Cancer data sets, obviously, there exists only a very slight difference between them.
- MGR algorithm outperforms MMR on all of four data sets, outperforms COOLCAT on three data sets, outperforms G-ANMI and k-ANMI on two data sets, and outperforms IG-ANMI on one data set.
- MGR improves the average clustering accuracy by 19% (from 0.696 to 0.83) as compared with MMR.
- IG-ANMI performs much better than other five algorithms on the Mushroom data set, which indicates IG-ANMI has the performance advantage on large data sets.
- The Zoo data set has unbalanced class distribution and the most number of clusters. An important observation is that MGR has the highest accuracy and IG-ANMI has the second highest accuracy on the Zoo data set, which shows the advantage of attribute-oriented clustering algorithms on such data sets.

- k-ANMI performs better than MGR and IG-ANMI on Votes and Cancer data sets, however, the algorithm is not stable enough. It can be observed that k-ANMI has the highest accuracy on Breast Cancer data set and the second highest accuracy on Votes data set while has the lowest accuracy on Zoo data set. Compared to k-ANMI, MGR and IG-ANMI have higher stability.

5.4 EFFICIENCY ANALYSIS

Ten large synthetically generated datasets R1, R2 through R10 are used to evaluate the efficiency of MGR and IG-ANMI algorithms. The running time of algorithms is used as the criteria for evaluation. G-ANMI is very time-consuming, for instance, it takes 20759 seconds to mine two clusters from Mushroom data set when the population size is set to 50. Thus this section mainly compares the running time of other five algorithms. The efficiencies of G-ANMI and IG-ANMI on four real-life data sets will be compared in Section 5.7. Five algorithms are sequentially applied to ten data sets. For fair comparison purpose, the average running time of each algorithm on a data set is obtained by using ten times run. The running time of COOLCAT varies with different percent of reprocess. Thus COOLCAT is run ten times with each percent of reprocess, and then the average running time is calculated. For IG-ANMI, the population size is set to 50. The running times in seconds of five algorithms on ten data sets are summarized in Table 5.39. Figure 5.2 illustrates the comparisons of running time among these five algorithms.

It can be seen from Table 5.39 and Figure 5.2, IG-ANMI takes the most time on all ten data sets and has the highest average running time, which indicates IG-ANMI has the lowest efficiency. Reversely, MGR algorithm takes the least time on all ten data sets and has the lowest average running time, which indicates MGR has the highest efficiency. The efficiency of MMR and COOLCAT are close to MGR. k-ANMI has the second highest average running time. Note that COOLCAT has the lower average running time than MMR. That is because the average running time of COOLCAT with different percent of reprocess is used. COOLCAT will have the higher average running time than MMR if only the running time of COOLCAT with 40% of reprocess is used.

Table 5.39: The running time in seconds of five algorithms on ten data sets

Algorithms	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	Average
MMR	2.836	5.672	8.992	12.453	17.547	22.117	27.109	31.93	38.219	43.102	20.998
MGR	0.805	1.656	2.649	3.782	5.141	6.711	8.376	10.196	12.367	15.024	6.671
k-ANMI	23.992	47.852	71.68	95.867	119.758	144.578	169.765	193.656	219.766	244.672	133.159
COOLCAT	1.918	3.695	5.77	7.609	9.906	12.059	14.254	15.778	18.922	20.992	11.09
IG-ANMI	78.688	143.375	219.25	295.453	457.796	437.25	509.375	591.843	671.657	756.954	416.164

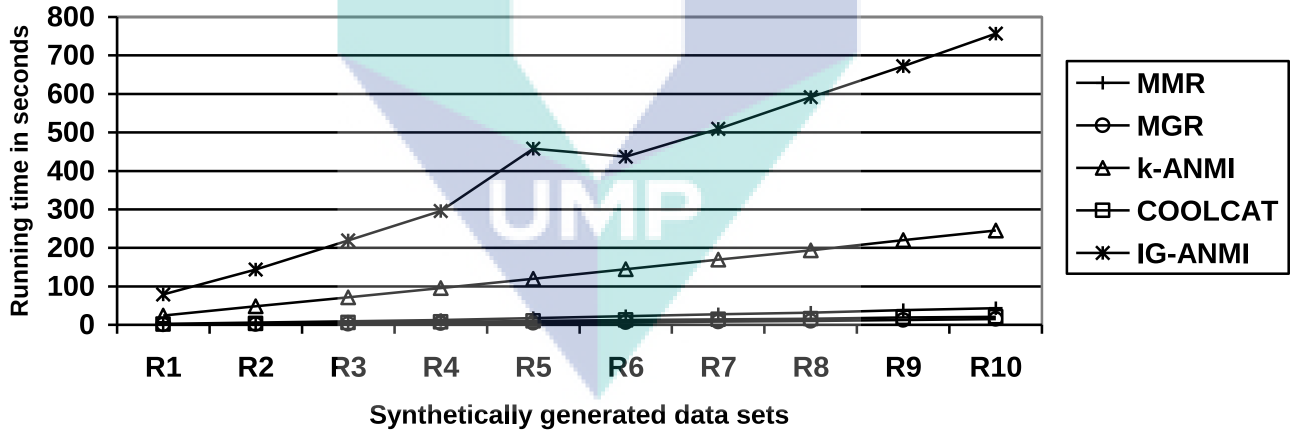


Figure 5.2: Running time of five algorithms on ten data sets

5.5 RUNNING MGR WITHOUT SPECIFYING THE NUMBER OF CLUSTERS

Different from other five algorithms, MGR algorithm can be run without specifying the desired number of clusters and end automatically. MGR algorithm is called Auto MGR for short in such case; correspondingly, the previous MGR algorithm is called Specified MGR for short. In this section, Auto MGR is applied on four real life data sets and ten synthetically generated data sets, respectively. Tables 5.40-5.43 show the clustering result of Auto MGR on four real life data sets. The accuracy values in these tables are obtained by using Eq. (5.1).

Table 5.40: The results of Auto MGR algorithm on the Zoo data set

No. of Cluster	Mammal	Fish	Bird	Invertebrate	Insect	Amphibian	Reptile	Max Number	Accuracy
1	41	0	0	0	0	0	0	41	
2	0	13	0	0	0	0	0	13	
3	0	0	20	0	0	0	0	20	
4	0	0	0	0	4	0	0	4	0.931
5	0	0	0	0	0	4	5	5	
6	0	0	0	7	0	0	0	7	
7	0	0	0	2	3	0	0	3	
8	0	0	0	1	1	0	0	1	

Table 5.41: The results of Auto MGR algorithm on the Votes data set

No. of Cluster	Votes	Republicans	Democrats	Max Number	Accuracy
1	208	8	200	200	
2	212	157	55	157	0.848
3	15	3	12	12	

Table 5.42: The results of Auto MGR algorithm on the Breast Cancer data set

No. of Cluster	Instances	Benign	Malignant	Max Number	Accuracy
1	373	369	4	369	0.93
2	45	37	8	37	
3	52	27	25	27	
4	27	0	27	27	
5	24	1	23	23	
6	29	7	22	22	
7	49	0	49	49	
8	84	3	81	81	

Table 5.43: The results of Auto MGR algorithm on the Mushroom data set

No. of Cluster	Instances	Poisonous	Edible	Max Number	Accuracy
1	1296	1296	0	1296	0.865
2	556	44	512	512	
3	1120	256	864	864	
4	2000	1760	240	1760	
5	2432	560	1872	1872	
6	288	0	288	288	
7	432	0	432	432	

Table 5.44 summarizes the number of clusters and accuracy produced by Auto MGR on four real life data sets, the real number of clusters, and the accuracy produced by Specified MGR (obtained from Table 5.38). It can be seen that the numbers of clusters on Zoo and Votes data sets are very close to the real numbers of clusters. Although the numbers of clusters on Cancer and Mushroom data sets are greater than the real numbers of clusters, they are at an acceptable level. Figure 5.3 illustrates the comparison of the accuracies produced by Auto MGR and Specified MGR, respectively. Table 5.44 and Figure 5.3 show that Auto MGR has higher accuracies on Vote, Cancer, and Mushroom data sets than Specified MGR. The accuracies of Specified MGR are improved by 2.4% (from 0.828 to 0.848) on Vote data set, by 5.2%

(from 0.884 to 0.93) on Cancer data set, and by 27.8% (from 0.677 to 0.865) on Mushroom data set.

Table 5.44: The summary of the results of Auto MGR on four real life data sets

Data sets	**Real number of clusters**	**Number of clusters**	**Accuracy of Auto MGR**	**Accuracy of Specified MGR**
Zoo	7	8	0.931	0.931
Vote	2	3	0.848	0.828
Cancer	2	8	0.93	0.884
Mushroom	2	7	0.865	0.677

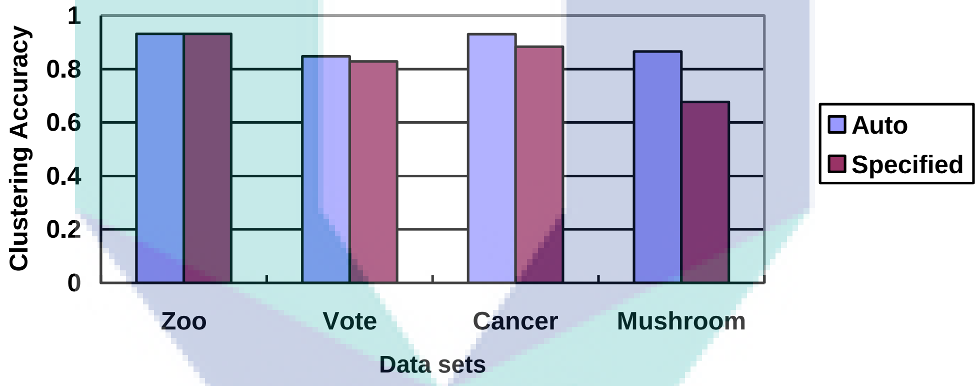


Figure 5.3: The comparison of the accuracies obtained by Auto MGR and Speicfied MGR

Table 5.45 summarizes the number of clusters and the running time in seconds produced by Auto MGR on ten synthetically generated data sets, the real number of clusters, and the running time taken by Specified MGR (obtained from Table 5.39). Figure 5.4 illustrates the comparison of the running times taken by Auto MGR and Specified MGR, respectively.

Table 5.45 shows that the numbers of clusters on all ten data sets are the same, i.e. 11. It is very close to the real numbers of clusters 10. Table 5.45 and Figure 5.4 show that the running time of Auto MGR on each data set is a little higher than that of Specified MGR. Such a little increase of running time is acceptable.

Table 5.45: The summary of the results of Auto MGR on R1-R10 data sets

Data sets	Real number of clusters	Number of clusters	Running time of Auto MGR	Running time of Specified MGR
R1	10	11	0.906	0.805
R2	10	11	1.703	1.656
R3	10	11	2.807	2.649
R4	10	11	3.968	3.782
R5	10	11	5.438	5.141
R6	10	11	6.99	6.711
R7	10	11	8.729	8.376
R8	10	11	10.714	10.196
R9	10	11	13.036	12.367
R10	10	11	15.87	15.024

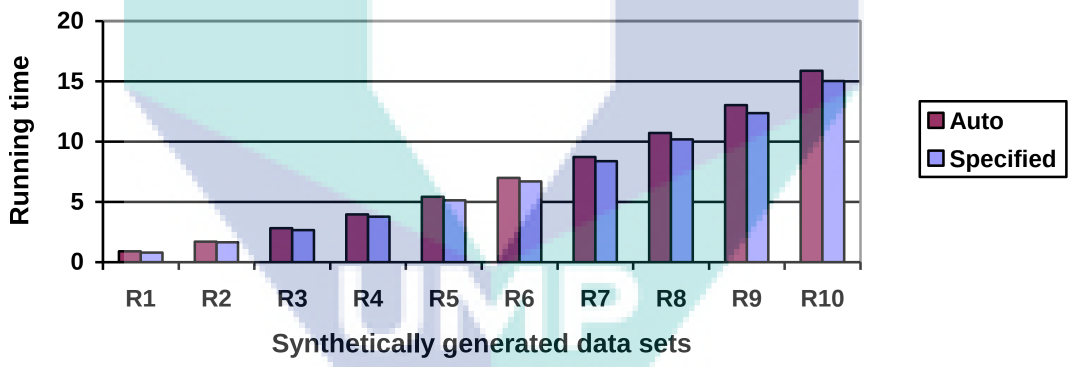


Figure 5.4: The comparison of the running times taken by Auto MGR and Specified MGR

5.6 SCALABILITY TEST

5.6.1 Scalability of MGR

The scalability of an algorithm refers to the ability of the algorithm for fitting the variances of the parameters related to data set or the algorithm itself. It is usually measured by the variance of running time of the algorithm with respect to some parameter. Two kinds of scalability of MGR algorithm are tested on ten synthetically generated data sets R1 through R10. One is the scalability with respect to these ten datasets when the number of clusters is fixed. The other is the scalability with respect to the number of clusters on dataset R10. Figure 5.5 shows the running time of using MGR program to find ten clusters in different datasets. Figure 5.6 shows the running time on R10 as number of clusters varies from 2 to 10.

It can be observed from Figure 5.5 that the running time of MGR algorithm tends to vary linearly with the increase of the number of objects, which is desired in practical applications. It can be observed from Figure 5.6 that the running times of MGR algorithm also vary linearly with respect to the number of clusters.

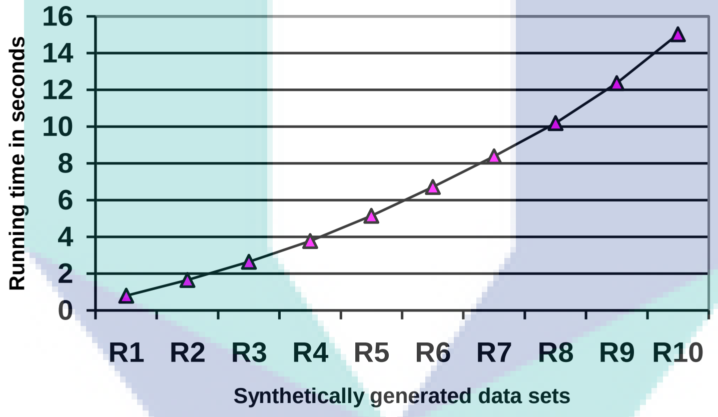


Figure 5.5: Scalability of MGR to the number of objects

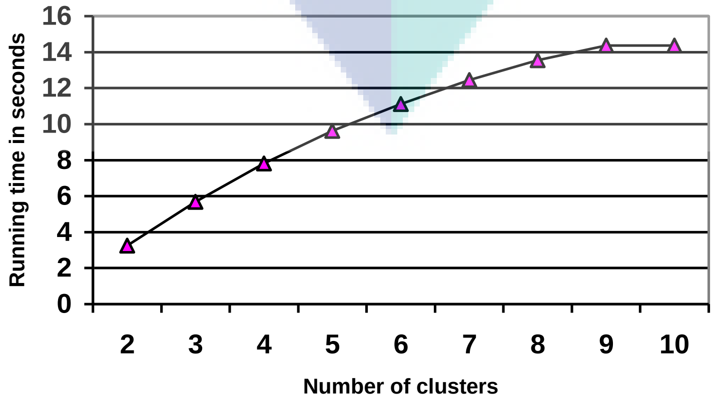


Figure 5.6: Scalability of MGR to the number of clusters

5.6.2 Scalability of IG-ANMI

Three kinds of scalability of IG-ANMI algorithm are tested on ten synthetically generated data sets and Mushroom data set. The first one is the scalability with respect to ten datasets R1-R10 when the number of clusters and population size are fixed, the second is the scalability with respect to the population size on Mushroom data set when the number of clusters is fixed, and the third is the scalability with respect to the number of clusters on Mushroom data set when population size is fixed. Figure 5.7 shows the running time of using IG-ANMI program to find two clusters from different datasets when population size is set to 50. Figure 5.8 shows the running time of mining two clusters on Mushroom as population size varies from 50 to 500. Given population size 50, Figure 5.9 shows the running time of IG-ANMI on Mushroom as number of clusters varies from 2 to 10.

It can be observed from Figure 5.7 that the running time of IG-ANMI algorithm tends to vary linearly with the increase of the number of objects, which is desired in practical applications. It can be observed from Figure 5.8 that the running times of IG-ANMI algorithm increase with the increase of population size. The value varies acutely at some population size. However, the scalability is at an acceptable level on the whole. From Figure 5.9, it can be seen that the running times of IG-ANMI maintain between interval [200, 300], which indicates the running times vary lightly with the increase of number of clusters.

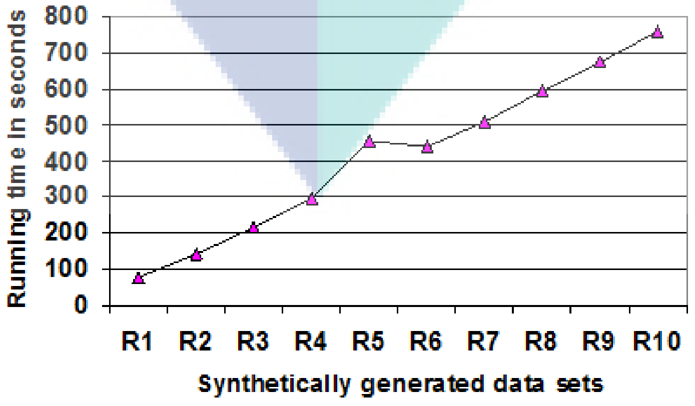


Figure 5.7: Scalability of IG-ANMI to the number of objects

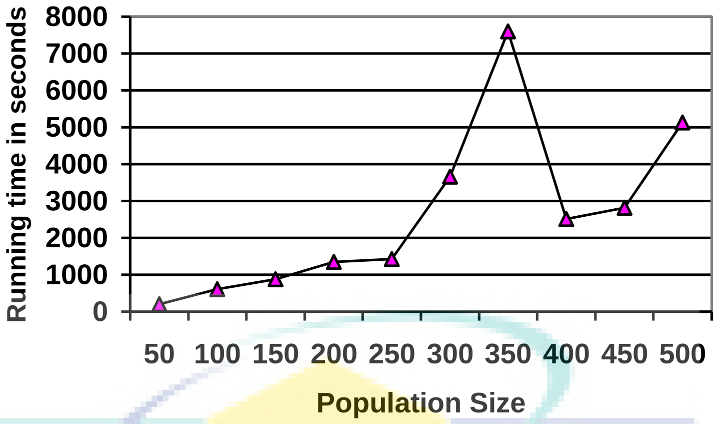


Figure 5.8: Scalability of IG-ANMI to the population size

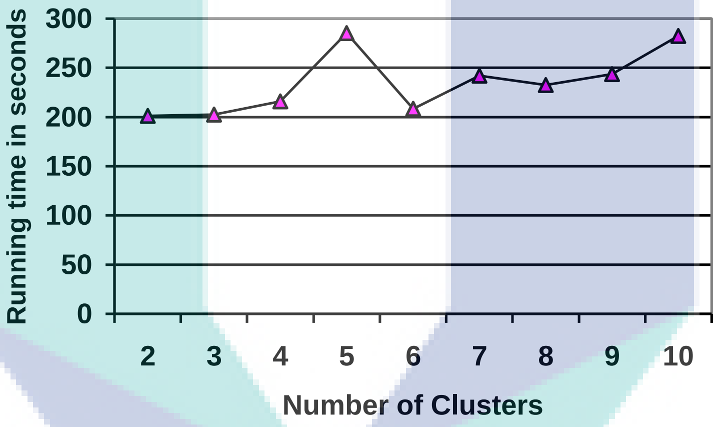


Figure 5.9: Scalability of IG-ANMI to the number of clusters

5.7 COMPARISON BETWEEN G-ANMI AND IG-ANMI

IG-ANMI improves G-ANMI by using a new initialization method. In order to appraise the influence of new initialization method on G-ANMI algorithm, the comparison between IG-ANMI and G-ANMI is performed on four real-life data sets.

5.7.1 Comparison on Efficiency

In this experiment, the number of iterations and running time of algorithms are used as the criteria for efficiency evaluation. Figures 5.10-5.13 plot the number of iterations of G-ANMI and IG-ANMI on four data sets as population size varies. It can be seen that IG-ANMI uses less iterations than G-ANMI. One exception is on the Zoo data set when population size is 500. On the Zoo, Votes, and Breast Cancer data sets, the smaller the population size is the bigger the difference between the numbers of iterations of these two algorithms. It is worth noting that there is a very large difference between G-ANMI and IG-ANMI on the Mushroom data set, which shows the greater advantage of IG-ANMI when larger data sets are processed.

Figures 5.14-5.17 plot the running time of G-ANMI and IG-ANMI on four data sets as population size varies. Since the running time is in proportion to the number of iterations, IG-ANMI takes less running time than G-ANMI except for on the Zoo data set when population size is 500. IG-ANMI greatly improves the efficiency of G-ANMI. Concretely, it is improved by 31% on the Zoo data set, 74% on the Votes data set, 59% on the Breast Cancer data set, and 3428% on the Mushroom data set. There is a very large difference between G-ANMI and IG-ANMI on the Mushroom data set, which indicates IG-ANMI can save much time when larger data sets are processed. Table 5.46 shows the concrete values of numbers of iterations and running time of G-ANMI and IG-ANMI on the Mushroom data set. When the population size is set to 200, G-ANMI takes 190998.485 seconds (53 hours) while IG-ANMI only take 1351.594 seconds (0.375 hour).

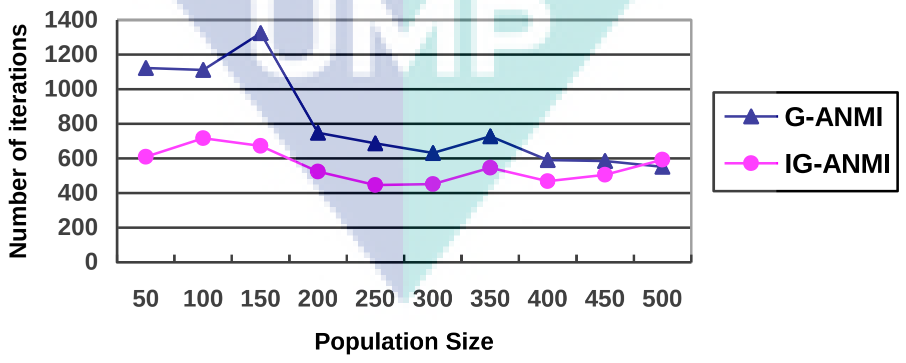


Figure 5.10: Number of iterations vs. population size on the Zoo data set

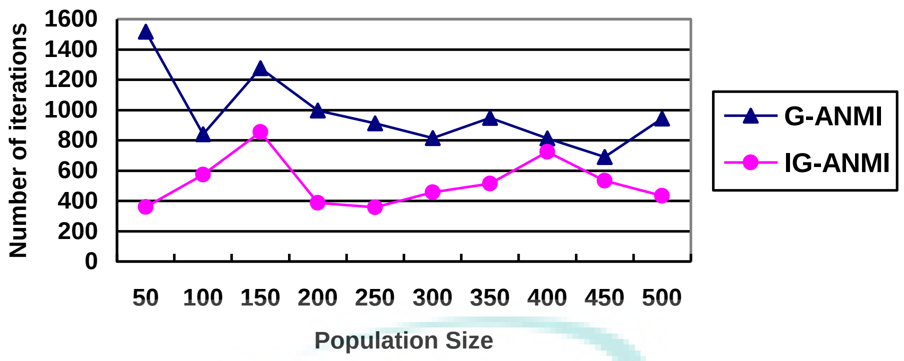


Figure 5.11: Number of iterations vs. population size on the Votes data set

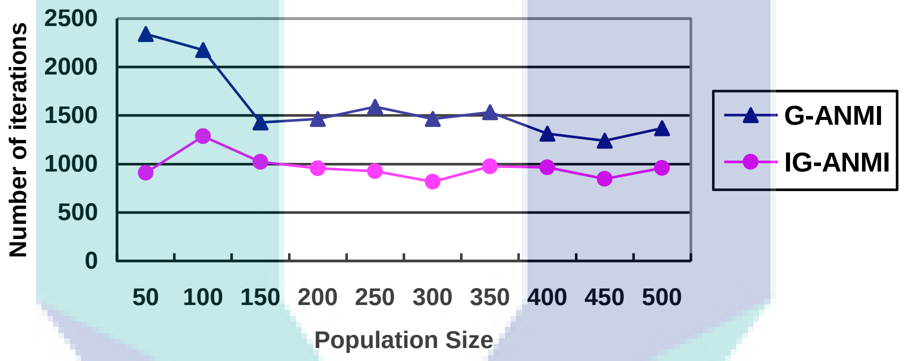


Figure 5.12: Number of iterations vs. population size on the Breast Cancer data set

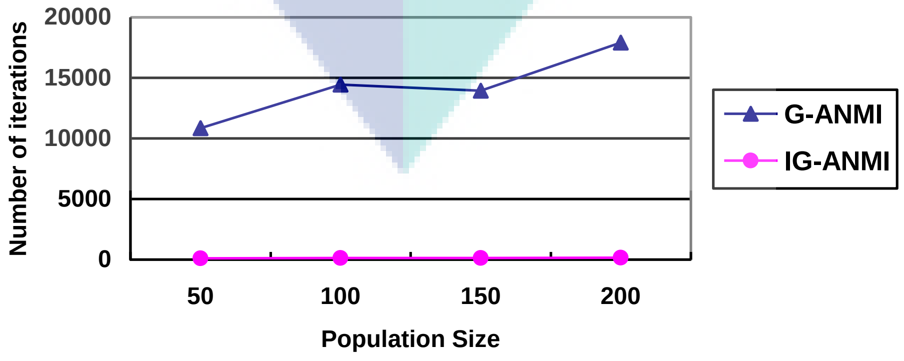


Figure 5.13: Number of iterations vs. population size on the Mushroom data set

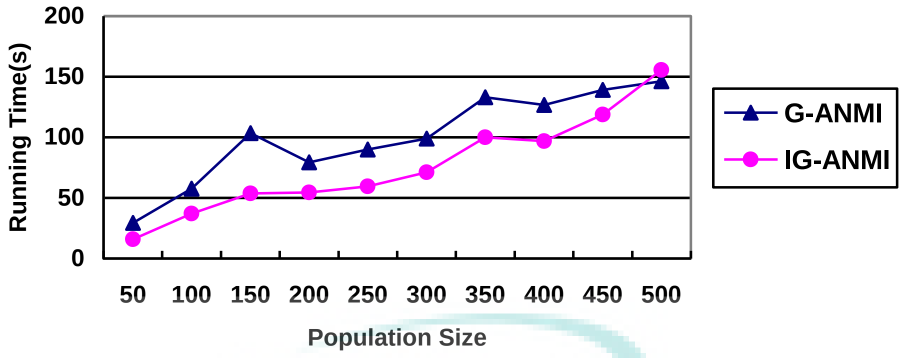


Figure 5.14: Running time vs. population size on the Zoo data set

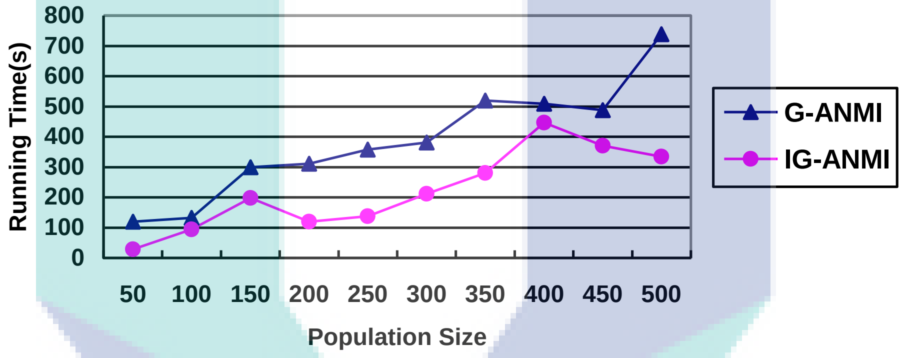


Figure 5.15: Running time vs. population size on the Votes data set

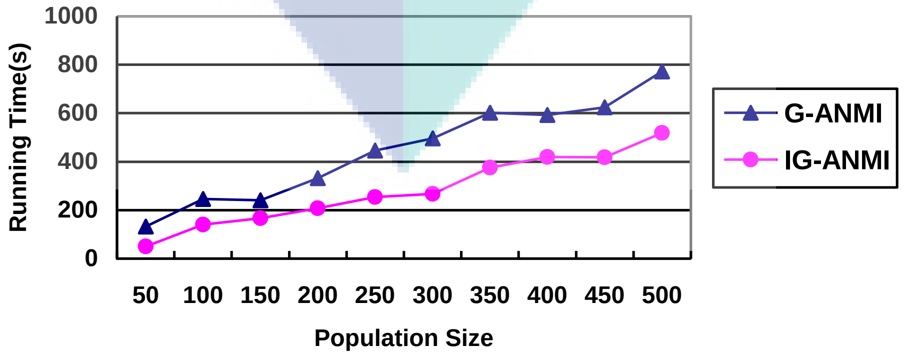


Figure 5.16: Running time vs. population size on the Breast Cancer data set

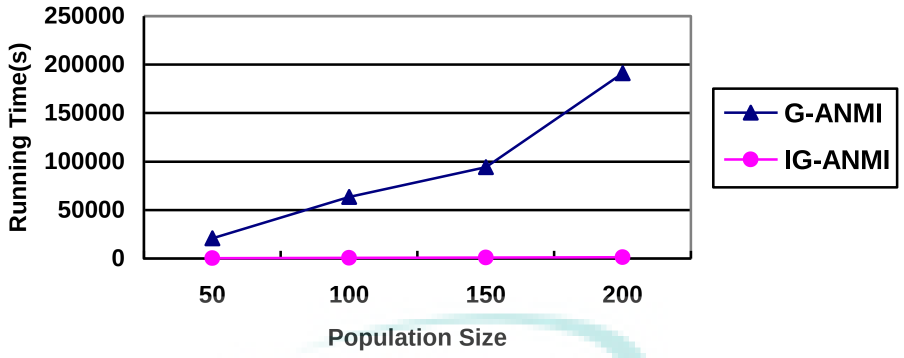


Figure 5.17: Running time vs. population size on the Mushroom data set

Table 5.46: The numbers of iterations and running time of G-ANMI and IG-ANMI on the Mushroom data set

Population Size	**Number of iterations**		**Running time (s)**	
	G-ANMI	**IG-ANMI**	**G-ANMI**	**IG-ANMI**
50	10845	100	20759.969	201.25
100	14453	145	63574.047	606.312
150	13944	144	94324.032	880.875
200	17916	158	190998.485	1351.594

5.7.2 Comparison on Performance

Clustering accuracy of algorithms is used as the criteria for performance evaluation. The clustering accuracies of these two algorithms on four real life data sets have been listed separately in Section 5.3. Table 5.47 summarizes the clustering accuracies of these two algorithms. From the average accuracies, it can be seen that IG-ANMI has higher clustering accuracy on the Zoo, Breast Cancer, and Mushroom data sets. One exception is on the Votes data set, the clustering accuracy of G-ANMI is slightly higher than that of IG-ANMI. The average clustering accuracy of G-ANMI on the four data sets is improved by 10.6%, from 0.815 of G-ANMI to 0.901 of IG-ANMI. It is worth noting that IG-ANMI improves clustering accuracy greatly on the Mushroom data set.

Table 5.47: The clustering accuracies of G-ANMI and IG-ANMI on four data sets

Data set	Algorithms	50	100	150	200	250	300	350	400	450	500	Average
Zoo	G-ANMI	0.832	0.851	0.911	0.881	0.861	0.891	0.931	0.812	0.851	0.921	0.874
	IG-ANMI	0.881	0.911	0.901	0.911	0.921	0.921	0.921	0.931	0.931	0.931	0.916
Votes	G-ANMI	0.874	0.862	0.876	0.864	0.880	0.874	0.869	0.864	0.876	0.874	0.871
	IG-ANMI	0.839	0.860	0.880	0.862	0.867	0.869	0.876	0.876	0.871	0.878	0.868
Breast Cancer	G-ANMI	0.924	0.962	0.960	0.963	0.972	0.975	0.975	0.975	0.977	0.978	0.966
	IG-ANMI	0.936	0.966	0.965	0.969	0.971	0.977	0.975	0.977	0.972	0.978	0.969
Mushroom	G-ANMI	0.547	0.568	0.546	0.538	-	-	-	-	-	-	0.55
	IG-ANMI	0.744	0.902	0.847	0.901	-	-	-	-	-	-	0.849

5.8 SUMMARY

A set of experiments are performed to appraise the clustering performance, clustering efficiency and scalability of MGR and IG-ANMI algorithms, and compare them with other four algorithms. The experimental results show that MGR improves the average clustering accuracy by 19% (from 0.696 to 0.83) as compared with MMR, at the same time maintains the highest efficiency among the existing algorithms for categorical data clustering. IG-ANMI greatly improves the efficiency of G-ANMI. Concretely, it is improved by 31% on the Zoo data set, 74% on the Votes data set, 59% on the Breast Cancer data set, and 3428% on the Mushroom data set. At the same time IG-ANMI maintains the highest clustering accuracy among the existing algorithms for categorical data clustering. IG-ANMI has obvious advantage against G-ANMI on large data sets in terms of clustering efficiency as well as clustering accuracy. MGR can be run without specifying the number of clusters (Auto MGR for short). The clustering accuracy produced by Auto MGR is higher than Specified MGR; the number of clusters produced by Auto MGR and the running time taken by Auto MGR are all at acceptable level. In addition, both of MGR and IG-ANMI have good scalability. The running time of MGR and IG-ANMI algorithms tend to vary linearly with the increase of the number of objects as well as the number of clusters.

CHAPTER 6

CONCLUSIONS

6.1 INTRODUCTION

This chapter consists of section 6.2 that summarizes the contributions and limitations of the thesis. Section 6.3 describes the future work based on the outcome of this thesis.

6.2 CONTRIBUTIONS AND LIMITATIONS

Corresponding to the objectives proposed in Chapter 1 (Section 1.3), the contributions of this research are summarized as follows:

- The limitations of algorithm MMR are investigated and a novel attribute-oriented hierarchical divisive clustering algorithm for categorical data termed MGR is developed. MGR overcomes the limitations of MMR using the information theory based concepts. MGR achieves higher clustering accuracy as compared with algorithm MMR (the average clustering accuracy on four real-life UCI data sets is improved by 19%, from 0.696 to 0.83), at the same time maintains the highest efficiency (the average running time on ten synthetic data sets is 6.671 seconds).
- The limitations of algorithm G-ANMI are investigated and an improved genetic clustering algorithm for categorical data termed IG-ANMI is developed. IG-ANMI algorithm improves G-ANMI by developing a new attribute-oriented initialization method. IG-ANMI greatly improves the efficiency of G-ANMI (improved by 31% on the Zoo data set, 74% on the

Votes data set, 59% on the Breast Cancer data set, and 3428% on the Mushroom data set) as well as the clustering accuracy of G-ANMI (the average clustering accuracy on four real-life UCI data sets is improved by 10.6%, from 0.815 to 0.901), at the same time maintains the highest clustering accuracy.

- One of the advantages of MGR is that it can be run without specifying the number of clusters. Clustering is an example of unsupervised learning, thus this is a better way than specifying the number of clusters especially when user experiences difficulties in specifying the number of clusters.
- The proposed algorithms MGR and IG-ANMI are evaluated on four real-life data sets obtained from UCI and ten synthetically generated data sets. Other four algorithms are used to compare with MGR and IG-ANMI algorithms. Experimental results show that both of MGR and IG-ANMI have good scalability, that is, they can be applied on small categorical data sets as well as large categorical data sets. They can be applied on the data sets which have balanced class distribution, such as Votes and Breast Cancer data sets, as well as the data sets which have unbalanced class distribution like Zoo data set. In addition, IG-ANMI has obvious advantage against G-ANMI on large data sets in terms of clustering efficiency as well as clustering accuracy.

To sum up, all the objectives proposed in Chapter 1 has been finished.

The proposed algorithms MGR and IG-ANMI produce good results. However, both of them have some of limitations as listed below:

- It has been pointed out that hierarchical clustering methods are not able to conduct adjustment when an opreation has been performed. In other words, if there are some problems in the process of clustering, these methods can not correct it (Han and Kamber, 2006). MGR also suffers from this problem.
- MGR may not generate good clusters when every attribute is uniformly distributed across all other attributes. In such case, the real clusters are not sufficiently distinguishable from each other. All the attributes have the very close (even the same) MGR values, however, the partitions they define are distinct. Therefore, it is possible that the equivalence classes in the partition

defined by the selected clustering attribute are dissimilar to the real clusters, which finally result in the clustering result is very dissimilar to the real clustering.

- The number of clusters is required to input in IG-ANMI. For user, however, sometimes it is difficult to know the number of clusters ahead.
- Both of MGR and IG-ANMI focus on categorical data clustering. At present, there also exist many databases with mixed numeric and categorical data. MGR and IG-ANMI cannot be applied on such databases.

6.3 FUTURE WORK

Further improvements can be made based on the outcomes from this thesis. Below are some possible future works:

- As mentioned in the last section, MGR algorithm is not able to conduct adjustment when an opreation has been performed. Therefore, a reprocess procedure like the iterative relocation process in the k-ANMI algorithm will be introduced to adjust the clustering results obtained by MGR algorithm for further improving the clustering accuracy.
- Experimental results show that the number of clusters obtained by MGR when it is run without specifying the number of clusters is usually greater than the real number of clusters. Therefore, a reprocess procedure will be introduced to combine some of the clusters produced by MGR. It can be implemented by first computing the similarity between the clusters based on a defined similarity measure, and then the clusters with the highest similarity are combined. Repeat these two steps until some conditions are satisfied.
- In the initial population produced by the initialization method of IG-ANMI, a part of chromosomes are generated by using the attributes partitions while the rest of chromosomes are generated randomly. More complicated initialization methods will be developed so that more equivalence classes of attributes partitions are integrated into the initial chromosomes. The efficiency of IG-ANMI is expected to be further improved by using such complicated initialization methods.

- Similar to most clustering algorithms for categorical data, IG-ANMI requires the number of clusters to be specified ahead. For user, however, sometimes it is difficult to know the number of clusters ahead. Thus, new approach will be developed to discover the number of clusters ahead and integrated into IG-ANMI.
- Recently, Particle Swarm Optimization (Kennedy and Eberhart, 1995; Kao et al., 2008) method has attracted much attention in the field of data mining. This method will be investigated and introduced into the problem of categorical data clustering.

REFERENCES

Andritsos, P. 2004. Scalable clustering of categorical data and applications, Ph.D. Dissertation. University of Toronto.

Andritsos, P., Tsaparas, P., Miller, R. and Sevcik, K. 2004. LIMBO: scalable clustering of categorical data, *Proceedings of 9th International Conference on Extending Database Technology*, pp. 123-146.

Atkinson-Abutridy, J., Mellish, C. and Aitken, S. 2004. Combining information extraction with genetic algorithms for text mining. *IEEE Intelligent Systems*. **19**(3): 22-30.

Bai, L., Liang, J.Y. and Dang, C.Y. 2011. An initialization method to simultaneously find initial cluster and the number of clusters for clustering categorical data, *Knowledge-Based Systems*. **24**: 785-795.

Barbara, D., Li, Y. and Couto, J. 2002. COOLCAT: an entropy-based algorithm for categorical clustering. *Proceedings of CIKM'02*, pp. 582-589.

Bobrowski, L. and Bezdek, J.C. 1991. c-Means clustering with the l_1 and l_∞ norms. *IEEE Transactions on Systems, Man and Cybernetics*. **21**(3): 545-554.

Cao, F.Y. and Liang, J.Y. 2011. A data labeling method for clustering categorical data. Expert Systems with Applications. **38**: 2381-2385.

Cao, F.Y., Liang, J.Y. and Bai, L. 2009. A new initialization method for categorical data clustering. Expert Systems with Applications. **33**(7): 10223-10228.

Cao, F.Y., Liang, J.Y., Li, D., Bai, L. and Dang, C. 2012. A dissimilarity measure for the k-Modes clustering algorithm. *Knowledge-Based Systems*. **26**: 120-127.

Chen, K. and Liu, L. 2005. The "best k" for entropy-based categorical data clustering. *Proceedings of International Conference on Scientific and Statistical Database Management*.

Chen, M. and Chuang, K. 2004. Clustering categorical data using the correlated-force ensemble. *Proceedings of SDM'04*.

Cheng, J., Qiao, M., Bian, W. and Tao, D. 2011. 3D human posture segmentation by spectral clustering with surface normal constraint. *Signal Processing*. **91**(9): 2204-2212.

Cristofor, D. and Simovici, D. 2002. Finding median partitions using information-theoretical-based genetic algorithms. *Journal of Universal Computer Science*. **8**(2): 153-172.

Deng, S., He, Z. and Xu, X. 2010. G-ANMI: A mutual information based genetic clustering algorithm for categorical data. *Knowledge-Based Systems*. **23**: 144-149.

Duntsch, I. and Gediga, G. 2000. *Rough set data analysis: A road to non-invasive knowledge discovery*. Bangor: Methodos.

Dutta, M., Mahanta, A.K. and Pujari, A. K. 2005. QROCK: A quick version of the ROCK algorithm for clustering of categorical data. *Pattern Recognition Letters*. **26**: 2364-2373.

Feng, S., Pang, J., Wang, D., Yu, G., Yang, F. and Xu, D. 2011. A novel approach for clustering sentiments in Chinese blogs based on graph similarity. *Computers & Mathematics with Applications*. **62**(7): 2770-2778.

Flake, G., Tarjan R. and Tsioutsiouliklis, k. 2004. Graph clustering and minimum cut trees. *Internet Mathematics*. **1**(4): 385-408.

Fu, T. 2011. A review on time series data mining. *Engineering Applications of Artificial Intelligence*. 24(1): 164-181.

Gan, G., Wu, J. and Yang, Z. 2009. A genetic fuzzy k-modes algorithm for clustering categorical data. *Expert Systems with Applications*. **36**: 1615-1620.

Ganti, V., Gehrke, J. and Ramakrishnan, R. 1999. CACTUS–clustering categorical data using summaries. *Proceedings of Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 73-83.

Giannotti, F., Gozzi, G. and Manco, G. 2002. Clustering transactional data. *Proceedings of PKDD'02*, pp. 175-187.

Gibson, D., Kleinberg, J. and Raghavan, P. 2000. Clustering categorical data: an approach based on dynamical systems. *The Very Large Data Bases Journal*. **8**(3–4): 222-236.

Gionis, A., Mannila, H. and Tsaparas, P. 2005. Clustering aggregation. *Proceedings of ICDE'05*, pp. 341-352.

Guha, S., Rastogi, R. and Shim, K. 2000. ROCK: a robust clustering algorithm for categorical attributes. *Information Systems*. **25** (5): 345-366.

Halkidi, M., Batistakis, Y. and Vazirgiannis, M. 2001. On clustering validation techniques. *Journal of Intelligent Information Systems*. **17**(2-3): 107-145.

Han, J. and Kamber, M. 2006. *Data mining: concepts and techniques*. 2nd ed. Amsterdam: Morgan Kaufmann.

Han, J., Kamber, M. and Pei, J. 2011. *Data mining: concepts and techniques*. 3rd ed. Morgan Kaufmann.

He, Z., Xu, X. and Deng, S. 2002. Squeezer: an efficient algorithm for clustering categorical data. *Journal of Computer Science & Technology*. **17**(5): 611-624.

He, Z., Xu, X. and Deng, S. 2005a. A cluster ensemble method for clustering categorical data. *Information Fusion*. **6**(2): 143-151.

He, Z., Xu, X. and Deng, S. 2005b. TCSOM: clustering transactions using self-organizing map. *Neural Processing Letters*. **22**(3): 249-262.

He, Z., Xu, X. and Deng, S. 2008. k-ANMI: a mutual information based clustering algorithm for categorical data. *Information Fusion*. **9**(2): 223-233.

He, Z., Xu, X. and Deng, S. 2011. Attribute value weighting in k-modes clustering. *Expert Systems with Applications*. **38**(12): 15365-15369.

Herawan, T., Deris, M.M. and Abawajy, J.H. 2010. A rough set approach for selecting clustering attribute. *Knowledge-Based Systems*. **23**: 220-231.

Holland, J.H. 1992. Adaptation in Natural and Artificial Systems. MIT Press.

Hu, X. 1995. Knowledge discovery in databases: an attribute oriented rough set approach. Ph.D. Dissertation. University of Regina.

Huang, R., Sang, N., Luo, D. and Tang, Q. 2011. Image segmentation via coherent clustering in L*a*b* color space. *Pattern Recognition Letters*. **32**(7): 891-902.

Huang, Z. 1997. A fast clustering algorithm to cluster very large categorical data sets in data mining, *Proceedings of 1997 SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pp. 1-8.

Huang, Z. 1998. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Mining and Knowledge Discovery*. **2**(3): 283-304.

Huang, Z. and Ng, M.K. 1999. A fuzzy k-modes algorithm for clustering categorical data. *IEEE Transaction on Fuzzy Systems*. **7**(4): 446-452.

Hung, C. and Peng, W. 2011. A regression-based approach for mining user movement patterns from random sample data. *Data & Knowledge Engineering*. **70**(1): 1-20.

Jain, A.K., Murty, M.N. and Flyn, P.J. 1999. Data clustering: a review. ACM Computing Surveys. 31(3): 264-323.

Jiang, D., Tang, C. and Zhang, A. 2004. Cluster analysis for gene expression data: a survey. *IEEE Transactions on Knowledge and Data Engineering*. **16**(11): 1370-1386.

Jollois, F. and Nadif, M. 2002. Clustering large categorical data. *Proceedings of PAKDD'02*, pp. 257-263.

Kalogeratos, A. and Likas, A. 2011. Document clustering using synthetic cluster prototypes. *Data & Knowledge Engineering*. **70**(3): 284-306.

Kalyani, S., Swarup, K.S. 2011. Particle swarm optimization based K-means clustering approach for security assessment in power systems. *Expert Systems with Applications*. **38**(9): 10839-10846.

Kao, Y., Zahara, E. and Kao, I. 2008. A hybridized approach to data clustering. *Expert Systems with Applications*. **34**(3): 1754-1762.

Kennedy, J. and Eberhart, R. C. 1995. Particle swarm optimization. *Proceedings of 1995 IEEE International Conference In Neural Networks, New Jersey, USA*, pp. 1942-1948.

Kim, D., Lee, K. and Lee, D. 2004. Fuzzy clustering of categorical data using fuzzy centroids. *Pattern Recognition Letters*. **25**(11): 1263-1271.

Liao, S., Chu, P. and Hsiao, P. 2012. Data mining techniques and applications-A decade review from 2000 to 2011. *Expert Systems with Applications*. http://dx.doi.org/ 10.1016/j.eswa.2012.02.063.

MacQueen, J. 1967. Some Methods for classification and Analysis of Multivariate Observations. *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*. **1**: 281-297.

Mali, K. and Mitra, S. 2003. Clustering and its validation in a symbolic framework. *Pattern Recognition Letters*. **24**(14): 2367-2376.

Man, K.F., Tang, K.S. and Kwong, S. 2001. Genetic Algorithms: Concepts and Designs. 3rd ed. Springer.

Mathieu, R. and Gibson, J. 2004. A methodology for large scale R&D planning based on cluster analysis. *IEEE Transactions on Engineering Management*. **40**(3): 283-292.

Mitchell, M. 1998. An introduction to genetic algorithms. MIT Press.

Ng, M.K. and Wong, J.C. 2002. Clustering categorical data sets using tabu search techniques. *Pattern Recognition*. **35**(12): 2783-2790.

Ngai, E.W.T., Hu, Y., Wong, Y.H., Chen, Y. and Sun, X. 2011. The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature. *Decision Support Systems*. **50**(3): 559-569.

Parmar, D., Wu, T. and Blackhurst, J. 2007. MMR: an algorithm for clustering categorical data using rough set theory. *Data and Knowledge Engineering*. **63**: 879-893.

Piatesky-Shapiro G., Fayyad, U. and Smyth, P. 1996. From data mining to knowledge discovery: an overview. *Advances in Knowledge Discovery and Data Mining*. AAA/MIT Press, 1-34.

Quinlan, J.R. 1986. Induction of decision trees. *Machine Learning*. **1**: 81-106.

Quinlan, J.R. 1993. C4.5: Programs for machine learning. Morgan Kaufmann.

Ralambondrainy, H. 1995. A conceptual version of the K-means algorithm. *Pattern Recognition Letters*. **16**(11): 1147–1157.

Rezaee, R., Lelieveldt, B.P.F. and Reiber, J.H.C. 1998. A new cluster validity index for the fuzzy c-mean. *Pattern Recognition Letters*. **19**: 237-246.

Saglam, B., Salman, F.S., Sayin, S. and Türkay, M. 2006. A mixed-integer programming approach to the clustering problem with an application in customer segmentation. *European Journal of Operational Research*. **173**(3): 866-879

San, O., Huynh V. and Nakamori, Y. 2004. An alternative extension of the k-means algorithm for clustering categorical data. *International Journal of Applied Mathematics, Computer Science*. **14**(2): 241-247.

Sever, H. 1998. The status of research on rough sets for knowledge discovery in databases. *Proceedings of the Second International Conference on Nonlinear Problems in Aviation and Aerospace*, **2**: 673-680.

Shi, J. and Malik, J. 2000. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. **22**(8): 888-905.

Slonim, N. and Tishby, N. 1999. Agglomerative information bottleneck. *In NIPS*, Breckenridge.

Strehl, A. and Ghosh, J. 2002. Cluster ensembles – a knowledge reuse framework for combining multiple partitions. *Journal of Machine Learning Research*. **3**: 583-617.

Sun, Y., Zhu, Q. and Chen, Z. 2002. An iterative initial-points refinement algorithm for categorical data clustering. *Pattern Recognition Letters*. **23**(7): 875-884.

Tan, P., Steinbach, M. and Kumar, V. 2006. *Introduction to data mining*. Boston: Addison Wesley.

UCI Machine Learning Repository. 2011. http://www.ics.uci.edu_/mlearn/ MLRepository.html.

Vretos, N., Solachidis, V. and Pitas, I. 2011. A mutual information based face clustering algorithm for movie content analysis. *Image and Vision Computing*. **29**(10): 693-705.

Wang, K., Xu, C. and Liu, B. 1999. Clustering transactions using large items, *Proceedings of CIKM'99*, pp. 483-490.

Wong, K., Feng, D., Meikle, S. and Fulham, M. 2002. Segmentation of dynamic pet images using cluster analysis. *IEEE Transactions on Nuclear Science*. **49**(1): 200-207.

Xu, J. and Sung, S.Y. 2003. Caucus-based transaction clustering. *Proceedings of DASFAA'03*, pp. 81-88.

Yun, C.H., Chuang, K.T. and Chen, M.S. 2001. An efficient clustering algorithm for market basket data based on small large ratios. *Proceedings of COMPSAC'01*, pp. 505-510.

Yun, C.H., Chuang, K.T. and Chen, M.S. 2002. Using category based adherence to cluster market-basket data. *Proceedings of ICDM'02*, pp. 546-553.

Zhao, P. and Zhang, C. 2011. A new clustering method and its application in social networks. *Pattern Recognition Letters*. **32**(15): 2109-2118.

APPENDIX A

PUBLICATIONS

The following publications had been made out of this thesis.

Journals

Hongwu Qin, Xiuqin Ma, Tutut Herawan, and Jasni Mohamad Zain, MGR: An information theory based hierarchical divisive clustering algorithm for categorical data, In submission for *Knowledge-Based Systems*.

Hongwu Qin, Xiuqin Ma, Tutut Herawan, and Jasni Mohamad Zain, An attribute-oriented initialization method for genetic clustering algorithm for categorical data, In submission for *Expert Systems with Applications*.

Hongwu Qin, Xiuqin Ma, Tutut Herawan, and Jasni Mohamad Zain, Mean information gain based approach for selecting clustering attribute, In submission for *International journal of data warehousing and mining*.

Conferences

Hongwu Qin, Xiuqin Ma, Jasni Mohamad Zain, Norrozila Sulaiman and Tutut Herawan, A mean mutual information based approach for selecting clustering attribute, *Communications in Computer and Information Science*, Volume 180, Software Engineering and Computer Systems, Part 1, Springer Verlag, Pages 1-15, 2011.

Hongwu Qin, Xiuqin Ma, Tutut Herawan, and Jasni Mohamad Zain, An improved genetic clustering algorithm for categorical data, Accepted for *the 2nd PAKDD Doctoral Symposium on Data Minning (DSDM'12)*, 29 May-1 June, 2012, Kuala Lumpur, Malaysia.

Hongwu Qin, Xiuqin Ma, Tutut Herawan, and Jasni Mohamad Zain, A novel attribute-oriented hierarchical divisive clustering algorithm for categorical data, Accepted for *the 3Clust workshop of PAKDD 2012*, 29 May-1 June, 2012, Kuala Lumpur, Malaysia.

APPENDIX B1

CODE LISTING FOR MGR ALGORITHM

There are six files in the source code of MGR.

1. header.h

```
# include "stdio.h"
# include "time.h"
# include "math.h"
# include "string.h"
# include "stdlib.h"
# define N  8        //Number of objects
# define M  7        //Number of attributes
# define MAX_LEN_OF_STR 20     //Max length of an attribute value
//Max size of the domain of an attribute
# define MAX_NUM_OF_EQUI_CLASS 15
# define MAX_NUM_OF_REAL_CLASS  15   //Max number of real clusters
# define PERCENT_THRESHOLD  0.03
```

2. main.c

```
# include "header.h"
int readdata_with_class(int data[][M], char filename[], int ObjClass[],
char ClassName[][MAX_LEN_OF_STR], int * NumOfClass);
void IGR_with_class(int OriginalData[][M], int RequNumOfCluster, int
ObjClass[], char ClassName[][MAX_LEN_OF_STR], int NumOfClass);

int main()
{
   int OriginalData[N][M];
   char filename[30];
   int RequNumOfCluster;  //Desired number of clusters
   int ObjClass[N];
   char ClassName[MAX_NUM_OF_REAL_CLASS][MAX_LEN_OF_STR];
   int NumOfClass;
   clock_t begin, end;
   double duration;

   printf("Please input the name of data file:");
   scanf("%s",filename);
   printf("\nPlease input the required number of clusters:");
   scanf("%d",&RequNumOfCluster);
   begin = clock();

   if(readdata_with_class(OriginalData, filename, ObjClass,
      ClassName,&NumOfClass)==0)
      exit(0);

   IGR_with_class(OriginalData, RequNumOfCluster, ObjClass,
                  ClassName,NumOfClass);
   end = clock();
   duration = (double)(end - begin)/CLOCKS_PER_SEC;
   printf("\nThe running time: %lf seconds\n",duration);
   getchar();
   return 0;
}
```

3. input.c

```
# include "header.h"
int readdata_with_class(int data[][M], char filename[], int
ObjClass[], char ClassName[][MAX_LEN_OF_STR], int * NumOfClass)
{
    int  n, m;
    int  i,j,k,s;
    int  cnt[M];  //Store the current number of value in a column
    char value_column[M][MAX_NUM_OF_EQUI_CLASS][MAX_LEN_OF_STR];
    char tempc;
    char tem[MAX_LEN_OF_STR];
    char pathin[50]= {".\\data\\"};
    char pathout[50]= {".\\data\\Cluster_Reslut\\"};

    strcat(strcat(pathin,filename),".data");
    if(freopen(pathin,"r", stdin)== NULL)
        return 0;
    strcat(strcat(pathout,filename),".out");
    if(freopen(pathout,"w", stdout)== NULL)
        return 0;
    scanf("%d,%d",&n,&m);
    getchar();
    for(j=0;j<m;j++)
       cnt[j] = 0;
    for(i=0;i<n;i++)
    {
           for(j=0;j<m;j++)
           {
                 k=0;
                 scanf("%c",&tempc);
                 while(tempc != ',' && tempc != 10 && tempc != '.')
                 {
                        tem[k]=tempc;
                        k++;
                        scanf("%c",&tempc);
                 }
                 tem[k]='\0';
                 //Finished reading an attribute value

                 if(i==0)
                 {
                         strcpy(value_column[j][0],tem);
                         cnt[j]++;
                         data[i][j]= 0;
                 }
                 else
                 {
                         for(s=0;s<cnt[j];s++)
                               if(strcmp(tem,value_column[j][s])==0)
                               {
                                     data[i][j] = s;
                                     break;
                        }
                if(s == cnt[j])
                {
                        strcpy(value_column[j][cnt[j]],tem);
                        data[i][j]= cnt[j] ;
                        cnt[j]++;
                }
        }
}
//Finished reading an object

//Read the class of the object
scanf("%c",&tempc);
if(tempc == '\'')
{
        k = 0;
        do
        {
                scanf("%c",&tempc);
                tem[k] = tempc;
                k++;
        }
        while(tempc != '\'');
        tem[k-1] = '\0';
        while(tempc != 10 && tempc != '.')
                scanf("%c",&tempc);
 }
 else
 {
         k = 0;
         do
         {
                 tem[k] = tempc;
                 k++;
                 scanf("%c",&tempc);
         }
         while(tempc != 10 && tempc != '.');
         tem[k] = '\0';
  }
  if(i==0)
  {
         strcpy(ClassName[0],tem);
         ObjClass[0]= 0;
         *NumOfClass = 1;
  }
  else
  {
         for(k= 0; k< *NumOfClass; k++)
                 if(strcmp(tem,ClassName[k])==0)
                 {
                         ObjClass[i]= k;
                         break;
                 }
         if(k == *NumOfClass)
         {
                 strcpy(ClassName[k],tem);
                 ObjClass[i]= k;
```

```
                                (*NumOfClass) ++;
                        }
                }
     }
     return 1;
}
```

4. my_algorithm.c

```
# include "header.h"
void GainOfDataset_ratio(int A[][M+1], int n, int m, int selectedAttr[]);
int EntropySplit(int selectedattr, int data[][M+1], int n, int
leafnode1[N+1], int leafnode2[N+1]);

void IGR_with_class(int OriginalData[][M], int RequNumOfCluster, int
ObjClass[], char ClassName[][MAX_LEN_OF_STR], int NumOfClass)
{
   int RemainData[N][M+1]; //Current data set
   int n,m;
   int i,j,k;
   int CurrNumOfCluster = 1;
   int selectedAttr[M];
   //Store the results of binary splitting
   int leafnode1[N+1],leafnode2[N+1];
   int num_of_each_real_class[MAX_NUM_OF_REAL_CLASS];
   float percent_of_each_real_class[MAX_NUM_OF_REAL_CLASS];
   int  max_num_of_each_real_class;
   int  total_max_num_of_each_real_class = 0;
   float accuracy;

   //Copy the initial data to current data
   for(i=0;i<N;i++)
   {
        for(j=0;j<M;j++)
              RemainData[i][j]=OriginalData[i][j];
        RemainData[i][M]= i;
   }
   n = N;
   m = M;

   while(CurrNumOfCluster < RequNumOfCluster)
   {
        GainOfDataset_ratio(RemainData, n, m, selectedAttr);
        for(i=0; i<M; i++)
        {
            if(EntropySplit(selectedAttr[i], RemainData, n,
               leafnode1, leafnode2)==1)
                  break;
        }
        if(i==M)
        {
            printf("Can not continue!!!\n");
            //Output current data as the last cluster
            for(j=0; j<MAX_NUM_OF_REAL_CLASS; j++)
            {
                  num_of_each_real_class[j]= 0;
                  percent_of_each_real_class[j]= 0.0;
        }
        printf("The %dth cluster has %d objects:",
                CurrNumOfCluster,n );
        for(j=0; j<n; j++)
        {
              k = ObjClass[RemainData[j][M]];
              num_of_each_real_class[k]++;
              printf("%d, ",RemainData[j][M]);
        }
        printf("where\n\n");
        max_num_of_each_real_class = 0;
        for(j=0; j<NumOfClass; j++)
        {
              if(num_of_each_real_class[j]>
                 max_num_of_each_real_class)
                 max_num_of_each_real_class =
                 num_of_each_real_class[j];
              percent_of_each_real_class[j]=
              (float)num_of_each_real_class[j]/(float)n;
              printf("%-5d  %-s  (%.1f%%)\n",
                     num_of_each_real_class[j],ClassName[j],
                     percent_of_each_real_class[j]*100 );
        }
        total_max_num_of_each_real_class +=
        max_num_of_each_real_class;
        accuracy =
        (float)total_max_num_of_each_real_class/(float)N;
        accuracy = (int)(accuracy*1000+0.5)/1000.0;
        printf("\n\nThe accuracy of clustering is:  %.1f%%",
               accuracy*100);
        return;
   }
   for(i=0;i<MAX_NUM_OF_REAL_CLASS;i++)
   {
        num_of_each_real_class[i]= 0;
        percent_of_each_real_class[i]= 0.0;
   }
   //Output leafnode1 as a cluster
   printf("The %dth cluster has %d objects:\n ",
          CurrNumOfCluster,leafnode1[N]);
   for(i=0; i<leafnode1[N]; i++)
   {
        j = ObjClass[leafnode1[i]];
        num_of_each_real_class[j]++;
        printf("%d, ",leafnode1[i]);
   }
   printf("where\n\n");
   max_num_of_each_real_class = 0;
   for(i=0; i<NumOfClass; i++)
   {
        if(num_of_each_real_class[i] > max_num_of_each_real_class)
           max_num_of_each_real_class = num_of_each_real_class[i];
        percent_of_each_real_class[i]=
        (float)num_of_each_real_class[i]/(float)leafnode1[N];
        printf("%-5d  %-s  (%.1f%%)\n",
               num_of_each_real_class[i],ClassName[i],
               percent_of_each_real_class[i]*100 );
        }
        total_max_num_of_each_real_class += max_num_of_each_real_class;
        printf("\n\n");
        CurrNumOfCluster++;

        //Rebuild current data using leafnode2
        for(i=0; i<leafnode2[N]; i++)
        {
             for(j=0; j<M; j++)
             {
                   RemainData[i][j] = OriginalData[leafnode2[i]][j];
             }
             RemainData[i][M] = leafnode2[i];
        }
        n = leafnode2[N];
    } //end of while

    //Output current data as the last cluster when the number of
    //clusters equals the desired number of clusters
    for(i=0; i<MAX_NUM_OF_REAL_CLASS; i++)
    {
         num_of_each_real_class[i]= 0;
         percent_of_each_real_class[i]= 0.0;
    }
    printf("The %dth cluster has %d objects: ", CurrNumOfCluster,n );
    for(i=0; i<n; i++)
    {
         j = ObjClass[RemainData[i][M]];
         num_of_each_real_class[j]++;
         printf("%d, ",RemainData[i][M]);
    }
    printf("where\n\n");
    max_num_of_each_real_class = 0;
    for(i=0; i<NumOfClass; i++)
    {
         if(num_of_each_real_class[i] > max_num_of_each_real_class)
              max_num_of_each_real_class = num_of_each_real_class[i];
         percent_of_each_real_class[i]=
         (float)num_of_each_real_class[i]/(float)n;
         printf("%-5d  %-s  (%.1f%%)\n",num_of_each_real_class[i],
                ClassName[i], percent_of_each_real_class[i]*100 );
    }
    total_max_num_of_each_real_class += max_num_of_each_real_class;
    accuracy = (float)total_max_num_of_each_real_class / (float)N;
    accuracy = (int)(accuracy*1000+0.5)/1000.0;
    printf("\n\nThe accuracy of clustering is:  %.1f%%", accuracy*100);
}
```

5. entropy_gainratio.c

```
# include "header.h"
void sort(float A[],int m, int order[]);

//Calculate the entropy of an attribute
float entropyA(int A[],int num)
{
   int i,j,classnumber;
```

```
    int numofeachclass[N+1]={0},valofeachclass[N+1]={0};
    float sum=0.0,temp;

    //Count the number of category in an attribute
    classnumber=1;
    valofeachclass[1]=A[0];
    numofeachclass[1]=1;
    for(i=1;i<num;i++)
    {
        for(j=1;j<=classnumber;j++)
             if (A[i]==valofeachclass[j])
             {
                   numofeachclass[j]++;
                   break;
             }
        if(j>classnumber)
        {
             classnumber++;
             numofeachclass[classnumber]++;
             valofeachclass[classnumber]=A[i];
        }
    }
    //Calculate the entropy
    for(i=1;i<=classnumber;i++)
    {
          temp=(float)numofeachclass[i]/(float)num;
          sum=sum+(-temp*log(temp)/log(2));
    }
    return sum;
}

//Calculate the entropy of an equivalence class of attribute B with
//respect to attribute A
static float entro_1ofB_A(int A[],int B[],int numoftheclass)
{
    int i,j;
    int classnumofB;
    int numofeachclassB[N+1]={0},valofeachclassB[N+1]={0};
    float sum=0.0,temp;

    classnumofB=1;
    valofeachclassB[1] = A[B[0]];
    numofeachclassB[1] = 1;
    for(i=1;i<numoftheclass;i++)
    {
        for(j=1;j<=classnumofB;j++)
             if (A[B[i]]==valofeachclassB[j])
             {
                   numofeachclassB[j]++;
                   break;
             }
        if(j>classnumofB)
        {
             classnumofB++;
             numofeachclassB[classnumofB]++;
             valofeachclassB[classnumofB]=A[B[i]];
        }
    }
    for(i=1;i<=classnumofB;i++)
    {
        temp=(float)numofeachclassB[i]/(float)numoftheclass;
        sum=sum+(-temp*log(temp)/log(2));
    }
    return sum;
}

static float entropyBA(int A[],int B[],int num)
{
    int i,j,k;
    int classnumofB;
    int numofeachclassB[N+1]={0},valofeachclassB[N+1]={0};
    int oneclassofB[N];
    float sum=0.0,temp;

    classnumofB=1;
    valofeachclassB[1]=B[0];
    numofeachclassB[1]=1;
    for(i=1;i<num;i++)
    {
        for(j=1;j<=classnumofB;j++)
             if(B[i]==valofeachclassB[j])
             {
                    numofeachclassB[j]++;
                    break;
             }
        if(j>classnumofB)
        {
             classnumofB++;
             numofeachclassB[classnumofB]++;
             valofeachclassB[classnumofB]=B[i];
        }
    }
    for(i=1;i<=classnumofB;i++)
    {
        k=0;
        for(j=0;j<num;j++)
             if(B[j]==valofeachclassB[i])
             {
                    oneclassofB[k]=j;
                    k++;
             }
        temp= entro_1ofB_A(A,oneclassofB,k);
        sum=sum+ (float)numofeachclassB[i]/(float)num *temp;
    }
    return sum;
}

void GainOfDataset_ratio(int A[][M+1], int n, int m, int
selectedAttr[])
{
    int i,j,k;
    int attrA[N]={0},attrB[N]={0};
    float entroA, sum, gain, temp_entropyBA;
    float gain_ratio[M];
    for(j=0;j<m;j++)
    {
         sum=0.0;
         for(i=0;i<n;i++)
             attrA[i]=A[i][j];
         entroA= entropyA(attrA,n);
         for(i=0;i<m;i++)
         {
             if(i!=j)
             {
                   for(k=0;k<n;k++)
                         attrB[k]=A[k][i];
                   temp_entropyBA = entropyBA(attrA,attrB,n);
                   sum=sum+ temp_entropyBA;
             }
         }
         gain=entroA*(m-1)-sum;
         if(fabs(entroA)>1e-6)
             gain_ratio[j] = gain/entroA;
         else
             gain_ratio[j] =0.0;
     }
     sort(gain_ratio, M, selectedAttr);
}
```

6. binary_split.c

```
# include "header.h"
float entropyA(int A[],int num);
static int elementinset(int e,int A[],int lenA);
static void sort(float A[],int m, int order[]);

int EntropySplit(int selectedattr, int data[][M+1], int n, int
leafnode1[N+1], int leafnode2[N+1])
{
     int i,j;
     int NumOfEquiClass;
     int EquiClasses[MAX_NUM_OF_EQUI_CLASS][N];
     int NumOfEachEquiClass[MAX_NUM_OF_EQUI_CLASS];
     float EntropyOfEachEquiClass[MAX_NUM_OF_EQUI_CLASS];
     int flag[N];
     int k1,k2;
     int order[MAX_NUM_OF_EQUI_CLASS];
     float percent;
     int k, tempA[N];
     float attrEntropy;

     NumOfEquiClass = 0;
     for(i=0;i<N;i++)
         flag[i]= -1;
     for(i=0;i<n;i++)
     {
         if(flag[i]==-1)
         {
              flag[i]=NumOfEquiClass;
              for(j=i+1;j<n;j++)
```

```
                {
                     if(data[j][selectedattr] == data[i][selectedattr])
                          flag[j]= NumOfEquiClass;
                }
                NumOfEquiClass++;
           }
      }
      for(i=0; i<NumOfEquiClass; i++)
      {
           NumOfEachEquiClass[i]=0;
           for(j=0;j<n;j++)
           {
                if(flag[j]==i)
                {
                     EquiClasses[i][NumOfEachEquiClass[i]]= j;
                     NumOfEachEquiClass[i]++;
                }
           }
           EntropyOfEachEquiClass[i] = 0;
           //Calculate the entropy of equivalence classes
           for(j=0; j<M; j++)
           {
                for(k = 0; k<NumOfEachEquiClass[i]; k++)
                     tempA[k]= data[EquiClasses[i][k]][j];

                attrEntropy = entropyA(tempA, NumOfEachEquiClass[i]);
                EntropyOfEachEquiClass[i] += attrEntropy;
           }

       }
       sort(EntropyOfEachEquiClass, NumOfEquiClass, order);
       for(i=0; i< NumOfEquiClass-1; i++)
       {
             percent = (float)NumOfEachEquiClass[order[i]]/(float)N;
             percent = (int)(percent*100+0.5)/100.0;
             if(percent - PERCENT_THRESHOLD > 1e-6)
                   break;
       }
       if(i == NumOfEquiClass-1)
             return 0;
       k1 = 0;
       k2 = 0;
       for(j=0; j<n; j++)
       {
             if(elementinset(j,EquiClasses[order[i]],
                               NumOfEachEquiClass[order[i]]))
             {
                   leafnode1[k1] =  data[j][M];
                   k1++;
             }
             else
             {
                   leafnode2[k2] =  data[j][M];
                   k2++;
             }
       }
       leafnode1[N] = k1;
```

```
        leafnode2[N] = k2;
        return 1;
}

static int elementinset(int e,int A[],int lenA)
{
        int i;
        for(i=0;i<lenA;i++)
            if(e==A[i])
                 return 1;
        return 0;
}

static void sort(float A[],int m, int order[])
{
        int i,j;
        float temp;

        for(i=0; i<m; i++)
              order[i] = i;
        for(i=1; i<m ;i++)
        {
              for(j=0;j<m-i;j++)
              {
                    if((A[j]-A[j+1])>1e-6)
                    {
                          temp = A[j+1];
                          A[j+1]= A[j];
                          A[j]= temp;
                          temp = order[j+1];
                          order[j+1]= order[j];
                          order[j]= temp;
                    }
              }
        }
}
```

APPENDIX B2

CODE LISTING FOR IG-ANMI ALGORITHM

There are seven files in the source code of IG-ANMI.

1. header.h

```
# include "stdio.h"
# include "string.h"
# include "stdlib.h"
# include "time.h"
# include "math.h"

# define N 10       //Number of objects
# define M 4        //Number of attributes
# define K 3        //Number of clusters
# define P 10       //Population size

# define CR 0.8     //Percent of chromes that are used for crossover
# define MU 0.1     //Percent of chromes that undergo a mutation
//Threshold for no relative improvement
# define IMPROVEMENT_THRESHOLD 1e-6
//Maximum number of consecutive iterations without improvement
# define MAX_ITERATION 100
# define MAX_LEN_OF_STR 20   //Max length of an attribute value
//Max size of domain of an attribute
# define MAX_NUM_OF_EQUI_CLASS 15
# define MAX_NUM_OF_REAL_CLASS 15  //Max number of real clusters

//The basic structure for computing ANMI
struct Node_of_histogram
{
      char category_value[MAX_LEN_OF_STR];
      int num;
};
```

2. g_anmi.c

```
# include "header.h"
void read_data(int, int cnt[M], struct Node_of_histogram
AH[][MAX_NUM_OF_EQUI_CLASS], int len_of_AH[], int ObjClass[], char
ClassName[][MAX_LEN_OF_STR], int * NumOfClass);
void initialization(int population[P][N], int data [N][M], int
num_of_eq_class[M]);
void compute_fitness(FILE * fp, int population[P][N], float fitness[P],
struct Node_of_histogram AH[][MAX_NUM_OF_EQUI_CLASS], int len_of_AH[]);
void generate_new_population(int population[P][N], float fitness[P]);
void output(int ObjClass[], char ClassName[][MAX_LEN_OF_STR], int NumOfClass,
int clusters[K][N], int len_of_clusters[]);

int main()
{
      FILE *   fp;
      int      i,j;
int     population[P][N];   //Store population with P chromes
float   fitness[P];         //Store fitness value
float   best_fitness_value;
int     id_of_best_fitness_value;
float   current_best_fitness_value;
int     cnt_without_improvement;
struct Node_of_histogram AH[M][MAX_NUM_OF_EQUI_CLASS];
int len_of_AH[M];
int clusters[K][N];      //Store K clusters
int len_of_clusters[K];  //Store the length of K clusters
int value[K];
int cnt;
//The following 3 variables are for computing accuracy
int ObjClass[N];      //The real classes of objects
char ClassName[MAX_NUM_OF_REAL_CLASS][MAX_LEN_OF_STR];
int NumOfClass;
char filename[30];
char path_data[50]= {".\\data\\"};
char path_result[50]= {".\\data\\Cluster_Result\\"};
//The following 3 variables are for computing running time
clock_t begin, end;
double duration;
int iteration_times = 0; //Store number of iterations
int data[N][M];    //Initial data (converted to integers)
int num_of_eq_class[M];    //Number of equivalence classes in
                           //each attribute
srand(1);     //Set random seed
//Open the input and output files
printf("Please input the name of data file:");
scanf("%s",filename);
begin = clock();  //Get initial time
strcat(strcat(path_data,filename),".data");
fp = freopen(path_data,"r", stdin);
strcat(strcat(path_result,filename),".out");
freopen(path_result,"w", stdout);

for(j=0;j<M;j++)
      num_of_eq_class[j] = 0;
//Read data from file, build histograms for attributes, and
//store real classes of objects.
read_data(data, num_of_eq_class, AH, len_of_AH, ObjClass,
          ClassName, &NumOfClass);

//Generate initial population
initialization(population, data, num_of_eq_class);
current_best_fitness_value = -1;
cnt_without_improvement = 0;

while(1)
{
      //compute fitness of each chromosome
      compute_fitness(fp, population, fitness, AH, len_of_AH);
      //search for the best fitness value
      best_fitness_value = fitness[0];
      id_of_best_fitness_value = 0;
      for(i=1; i<P; i++)
      {
```

```
                if(fitness[i] - best_fitness_value > 1e-6)
                {
                        best_fitness_value = fitness[i];
                        id_of_best_fitness_value = i;
                }
        }
        //There exists improvement on the best fitness value
        if(best_fitness_value - current_best_fitness_value >
            IMPROVEMENT_THRESHOLD)
        {
                current_best_fitness_value = best_fitness_value;
                cnt_without_improvement = 0;
        }
        else
                cnt_without_improvement ++;
        if(cnt_without_improvement >= MAX_ITERATION)
        {
                for(i=0; i<K; i++)
                        len_of_clusters[i] = 0;
                value[0]= population[id_of_best_fitness_value][0];
                cnt = 1;
                for(i=1; i<N; i++)
                {
                        for(j = 0; j<cnt; j++)
                          if(population[id_of_best_fitness_value][i]
                              == value[j])
                               break;
                        if(j == cnt)
                        {
                          value[cnt]=
                          population[id_of_best_fitness_value][i];
                          cnt++;
                        }
                        if(cnt == K)
                          break;
                }
                for(i=0; i<K; i++)
                {
                        for(j=0; j<N; j++)
                        {
                          if(population[id_of_best_fitness_value][j]
                              == value[i])
                          {
                              clusters[i][len_of_clusters[i]] = j;
                              len_of_clusters[i] ++;
                          }
                        }
                }
                output(ObjClass, ClassName, NumOfClass, clusters,
                        len_of_clusters);
                break;
        }
        generate_new_population(population, fitness);
        iteration_times++;
}
printf("\nIteration times: %d\n", iteration_times);
end = clock();
```

```
      duration = (double)(end - begin)/CLOCKS_PER_SEC;
      printf("\nThe running time: %lf seconds\n",duration);
      getchar();
      return 0;
}

void output(int ObjClass[], char ClassName[][MAX_LEN_OF_STR], int
            NumOfClass, int clusters[K][N], int len_of_clusters[])
{
      int i,j,s;
      int num_of_each_real_class[MAX_NUM_OF_REAL_CLASS];
      float percent_of_each_real_class[MAX_NUM_OF_REAL_CLASS];
      int  max_num_of_each_real_class;
      int  total_max_num_of_each_real_class = 0;     Í
      float accuracy;

      for(i=0; i<K; i++)
      {
            for(j=0; j<MAX_NUM_OF_REAL_CLASS; j++)
            {
                  num_of_each_real_class[j]= 0;
                  percent_of_each_real_class[j]= 0.0;
            }
            printf("\n\nThe %dth cluster has %d objects: ",
                   i,len_of_clusters[i]);
            for(j=0; j<len_of_clusters[i]; j++)
            {
                  s = ObjClass[clusters[i][j]];
                  num_of_each_real_class[s]++;
                  printf("%d, ",clusters[i][j]);
            }
            printf("where\n\n");
            max_num_of_each_real_class = 0;
            for(j=0; j<NumOfClass; j++)
            {
                  if(num_of_each_real_class[j]>
                     max_num_of_each_real_class)
                     max_num_of_each_real_class=
                     num_of_each_real_class[j];
                  percent_of_each_real_class[j]=
                  (float)num_of_each_real_class[j]/
                  (float)len_of_clusters[i];
                  printf("%-5d  %-s (%.1f%%)\n",
                         num_of_each_real_class[j],ClassName[j],
                         percent_of_each_real_class[j]*100 );
            }
            total_max_num_of_each_real_class +=
            max_num_of_each_real_class;
      }
      accuracy = (float)total_max_num_of_each_real_class / (float)N;
      accuracy = (int)(accuracy*1000+0.5)/1000.0;
      printf("\n\nThe accuracy of clustering is:
             %.1f%%",accuracy*100);
}
```

3. read_data.c

```
# include "header.h"

// This function reads data from file, creats histograms for
// attributes, and store the real classes of objects
void read_data(int data [N][M], int cnt[M], struct Node_of_histogram
       AH[][MAX_NUM_OF_EQUI_CLASS], int len_of_AH[], int ObjClass[],
       char ClassName[][MAX_LEN_OF_STR], int * NumOfClass)
{
      int obj_i,i,j,k;
      char tempc;
      char tem[MAX_LEN_OF_STR];
      char object[M][MAX_LEN_OF_STR];
      char value_column[M][MAX_NUM_OF_EQUI_CLASS][MAX_LEN_OF_STR];
      int s;

      for(i=0; i<M; i++)
            len_of_AH[i] = 0;
      for(obj_i=0; obj_i<N; obj_i++)
      {
            for(i=0; i<M; i++)
            {
                  j=0;
                  scanf("%c",&tempc);
                  while(tempc != ',' && tempc != 10 && tempc != '.')
                  {
                        object[i][j]=tempc;
                        j++;
                        scanf("%c",&tempc);
                  }
                  object[i][j]='\0';
                  if(obj_i==0)
                  {
                        strcpy(value_column[i][0], object[i]);
                        cnt[i]++;
                        data[obj_i][i]= 0;
                  }
                  else
                  {
                        for(s=0;s<cnt[i];s++)
                          if(strcmp(object[i],value_column[i][s])==0)
                          {
                              data[obj_i][i] = s;
                              break;
                          }
                        if(s == cnt[i])
                        {
                          strcpy(value_column[i][cnt[i]],object[i]);
                          data[obj_i][i]= cnt[i] ;
                          cnt[i]++;
                        }
                  }
            }
            scanf("%c",&tempc);
            if(tempc == '\'')
            {
```

```
            k = 0;
            do
            {
                  scanf("%c",&tempc);
                  tem[k] = tempc;
                  k++;
            }
            while(tempc != '\'');
            tem[k-1] = '\0';
            while(tempc != 10 && tempc != '.')
                  scanf("%c",&tempc);
      }
      else
      {
            k = 0;
            do
            {
                  tem[k] = tempc;
                  k++;
                  scanf("%c",&tempc);
            }
            while(tempc != 10 && tempc != '.');
                  tem[k] = '\0';
       }
       if(obj_i==0)
       {
            strcpy(ClassName[0],tem);
            ObjClass[0]= 0;
            *NumOfClass = 1;
       }
       else
       {
            for(k = 0; k< *NumOfClass; k++)
                  if(strcmp(tem,ClassName[k])==0)
                  {
                        ObjClass[obj_i]= k;
                        break;
                  }
            if(k == * NumOfClass)
            {
                  strcpy(ClassName[k],tem);
                  ObjClass[obj_i]= k;
                  (* NumOfClass) ++;
            }
       }
       //Build or update Histogram for each attribute
       for(i=0; i<M; i++)
       {
            for(j=0; j<len_of_AH[i]; j++)
              if(strcmp(object[i],AH[i][j].category_value)==0)
              {
                  AH[i][j].num++;
                  break;
              }
            if(j == len_of_AH[i])
            {
                  strcpy(AH[i][j].category_value,object[i]);
```

```
                            AH[i][j].num = 1;
                            len_of_AH[i]++;
                      }
                }
         } //end for(obj_i=0; obj_i<N; obj_i++)
}
```

4. initialization.c

```
# include "header.h"
//Generate the initial population including P chroms
void initialization(int population[P][N], int data [N][M], int
                    num_of_eq_class[M])
{
        int i,j,temp;
        int flag[K];
        int partition_tmp[N];
        int tmp[N];
        int tmp_flag[K];
        int s,q;
        int sum;

        for(i=0; i<M; i++)
        {
             for(j=0; j<N; j++)
                  partition_tmp[j] = -1;
             if(num_of_eq_class[i] == K)
             {
                  for(j=0; j<N; j++)
                        partition_tmp[j] = data[j][i];
             }
             else
             {
                  if(num_of_eq_class[i] > K)
                  {
                        for(j=0; j<N; j++)
                        {
                              if(data[j][i] < K)
                                    partition_tmp[j] = data[j][i];
                        }
                        for(j=0; j<N; j++)
                        {
                              if(partition_tmp[j] == -1)
                                    partition_tmp[j] = rand()% K;
                        }
                  }
                  else
                  {
                        q = num_of_eq_class[i];
                        for(j=q-2; j>=0; j--)
                        {
                              sum = 0;
                              for(s=0; s<N; s++)
                                    if(data[s][i] <= j)
                                          sum += 1;
                              if((N-sum) >= (K-j-1))
                                    break;
```

```
                }
                if(j == -1)
                {
                      while(1)
                      {
                            for(s=0; s<K; s++)
                                  flag[s] = 0;
                            for(s=0; s<N; s++)
                            {
                                  temp = rand()% K;
                                  flag[temp] = 1;
                                  partition_tmp[s]= temp;
                            }
                            for(s=0; s<K; s++)
                                  if(flag[s]==0)
                                        break;
                            if(s == K)
                                  break;
                      }
                }
                else
                {
                      for(s=0; s<N; s++)
                      {
                            if(data[s][i] <= j)
                               partition_tmp[s] = data[s][i];
                      }
                      while(1)
                      {
                            for(s=0; s<(K-j-1); s++)
                                  tmp_flag[s] = 0;
                            for(s=0; s<N-sum; s++)
                            {
                                  tmp[s] = rand()%(K-j-1);
                                  tmp_flag[tmp[s]] = 1;
                                  tmp[s]= tmp[s]+(j+1);
                            }
                            for(s=0; s<(K-j-1); s++)
                                  if(tmp_flag[s]==0)
                                        break;
                            if(s == (K-j-1))
                                  break;
                      }
                      q=0;
                      for(s=0; s<N; s++)
                            if(partition_tmp[s] == -1)
                            {
                                  partition_tmp[s] = tmp[q];
                                  q++;
                            }
                } //end else
          } //end else
      } // end else
      for(j=0; j<N; j++)
          population[i][j] = partition_tmp[j];
} //end for(i=0; i<M; i++)
```

```
        //Randomly generate P-M chroms
        for(i=M; i<P; )
        {
            for(j=0; j<K; j++)
                  flag[j] = 0;
            for(j=0; j<N; j++)
            {
                  temp = rand()% K;
                  flag[temp] = 1;
                  population[i][j]= temp;
            }
            for(j=0; j<K; j++)
                  if(flag[j]==0)
                        break;
            if(j == K)
                  i++;
         }
}
```

5. fitness.c

```
# include "header.h"
float Compute_ANMI(struct Node_of_histogram AH[][MAX_NUM_OF_EQUI_CLASS], int
len_of_AH[], struct Node_of_histogram CAH[K][M][MAX_NUM_OF_EQUI_CLASS], int
len_of_CAH[][M], int len_of_clusters[]);

void compute_fitness(FILE * fp, int population[P][N], float
fitness[P], struct Node_of_histogram AH[][MAX_NUM_OF_EQUI_CLASS], int
len_of_AH[])
{
        int i,j,k,obj_i;
        char tempc;
        char object[M][MAX_LEN_OF_STR];
        //Histograms for K clusters
        struct Node_of_histogram CAH[K][M][MAX_NUM_OF_EQUI_CLASS];
        int len_of_CAH[K][M];
        int id;

        int len_of_clusters[K];
        //compute fitness for each chrom
        for(i=0; i<P; i++)
        {
            //set the length of clusters and CAH 0
            for(j=0; j<K; j++)
            {
                  len_of_clusters[j] = 0;
                  for(k=0; k<M; k++)
                        len_of_CAH[j][k] = 0;
            }
            rewind(fp);
            for(obj_i=0; obj_i<N; obj_i++)
            {
                  //Read an object
                  for(j=0; j<M; j++)
                  {
                        k=0;
                        scanf("%c",&tempc);
                        while(tempc!=','&& tempc!= 10&& tempc!= '.')
                        {
                              object[j][k]=tempc;
                              k++;
                              scanf("%c",&tempc);
                        }
                        object[j][k]='\0';
                  }
                  while(tempc != 10 && tempc != '.')
                        scanf("%c",&tempc);
                  //Build Histograms
                  id = population[i][obj_i];
                  len_of_clusters[id]++;
                  for(j=0; j<M; j++)
                  {
                        for(k=0; k<len_of_CAH[id][j]; k++)
                             if(strcmp(object[j],
                             CAH[id][j][k].category_value)==0)
                             {
                                  CAH[id][j][k].num++;
                                  break;
                             }
                        if(k == len_of_CAH[id][j])
                        {
                             strcpy(CAH[id][j][k].category_value,
                             object[j]);
                             CAH[id][j][k].num = 1;
                             len_of_CAH[id][j]++;
                        }
                  }
            } //for(obj_i=0; obj_i<N; obj_i++)
            fitness[i] = Compute_ANMI(AH, len_of_AH, CAH, len_of_CAH,
                                      len_of_clusters);
      }//for(i=0; i<P; i++)
}
```

6. compute_anmi.c

```
# include "header.h"
float Compute_ANMI(struct Node_of_histogram AH[][MAX_NUM_OF_EQUI_CLASS], int
len_of_AH[], struct Node_of_histogram CAH[K][M][MAX_NUM_OF_EQUI_CLASS], int
len_of_CAH[][M], int len_of_clusters[])
{
        int b,h,g;
        int i;
        int ngh;
        int nh;
        int ng;
        float sum = 0.0;
        float sh;
        float sg;
        float NMI;
        float log_tmp1,log_tmp2;
        char str_tmp[MAX_LEN_OF_STR];
      //Compute ANMI between a partition and M attributes partitions
        for(b=0; b<M; b++)
        {
            sh = 0.0;
            for(h = 0; h<K; h++)        //k(a)=K
            {
                  sg = 0.0;
                  for(g=0; g< len_of_AH[b]; g++) //k(b)=len_of_AH[b]
                  {
                        //Compute ng(h)
                        strcpy(str_tmp, AH[b][g].category_value);
                        for(i=0; i<len_of_CAH[h][b]; i++)
                           if(strcmp(str_tmp,
                              CAH[h][b][i].category_value)==0)
                           {
                              ngh = CAH[h][b][i].num;
                              break;
                           }
                        if(i == len_of_CAH[h][b])
                           ngh = 0;
                        if(ngh != 0)
                        {
                           nh = len_of_clusters[h];  //n(h)
                           ng = AH[b][g].num;  //ng
                           log_tmp1 = (float)(ngh*N)/(float)(nh*ng);
                           log_tmp2 = (float)(K*len_of_AH[b]);
                           sg+= ngh * (log(log_tmp1)/log(log_tmp2));
                        }
                  }
                  sh = sh + sg;
            }
            //NMI between a partition and attribute b
            NMI = ((float)2/(float)N) * sh;
            sum = sum + NMI;
        }
        //Return ANMI
        return sum/M;
}
```

7. generate_new_population.c

```
# include "header.h"
void sort(float A[],int m, int order[]);

void generate_new_population(int population[P][N], float fitness[P])
{
        int i,j;
        int new_population[P][N];
        float fitness_copy[P];
        int order[P];
        float temp;
        int num_of_fittest;
        int num_of_crossover;
        //Probability of being selected of a chrom
        float selection_prob[P];
        float sum_of_fitness = 0.0;
        float rand_num;   //A number between [0,1]
```

```
int index[P];
int id;
int id1,id2;
int flag[P];
int k;
int crosspoint;
int num_of_mutation;
int num_of_mutation_point;
int flag_mutation_point[N];

for(i=0; i<P; i++)
    fitness_copy[i] = fitness[i];
sort(fitness_copy, P, order);
temp = (1-CR-MU)*P;
num_of_fittest = floor(temp);
//copy fittest (1-r-m)M chroms to new population
for(i=0; i<num_of_fittest; i++)
{
    for(j=0; j<N; j++)
        {
            new_population[i][j] = population[order[i]][j];
        }
}
//crossover
for(i=0; i<P; i++)
    sum_of_fitness += fitness[i];
for(i=0; i<P; i++)
    selection_prob[i] = fitness[i]/sum_of_fitness;
for(i=1; i<P; i++)
    selection_prob[i]=selection_prob[i-1]+selection_prob[i];
//Roulette wheel algorithm
temp = CR*P;
num_of_crossover = floor(temp);
for(i=0; i<num_of_crossover; i++)
{
    rand_num = rand()%1000/1000.0;
    id = 0;
    while(rand_num - selection_prob[id] > 1e-6)
          id++;
    index[i] = id;
}
k = num_of_fittest;
for(i=0; i<num_of_crossover; i++)
    flag[i] = 0;
for(i=0; i<num_of_crossover/2; i++)
{
    //Randomly select id1 and id2 to perform crossover
    do
    {
          id1 = rand()% num_of_crossover;
    }
    while(flag[id1]==1);
    flag[id1] = 1;
    do
    {
          id2 = rand()% num_of_crossover;
    }
    while(flag[id2]==1);
    flag[id2] = 1;
    crosspoint = rand()% N;
    for(j=0; j<=crosspoint; j++)
    {
          new_population[k][j]= population[index[id1]][j];
          new_population[k+1][j]= population[index[id2]][j];
    }
    for(j=crosspoint+1; j<N; j++)
    {
          new_population[k][j]= population[index[id2]][j];
          new_population[k+1][j]= population[index[id1]][j];
    }
    k = k+2;
}

//mutation
for(i=0; i<P; i++)
    flag[i] = 0;
temp = MU*P;
num_of_mutation = floor(temp);
temp = 0.1*N;
num_of_mutation_point = floor(temp);
if(num_of_mutation_point == 0)
    num_of_mutation_point = 1;
for(i=0; i<num_of_mutation; i++)
{
    do
    {
          id = rand()%P;
    }
    while(flag[id]==1);
    flag[id] = 1;
    for(j=0; j<N; j++)
          new_population[k][j] = population[id][j];
    for(j=0; j<N; j++)
          flag_mutation_point[j] = 0;
    for(j=0; j<num_of_mutation_point; j++)
    {
          do
          {
                id1 = rand()%N;
          }
          while(flag_mutation_point[id1]==1);
          flag_mutation_point[id1] = 1;
          new_population[k][id1] = rand()% K;
    }
    k++;
}

//Copy newpopulation to population
for(i=0; i<P; i++)
{
    for(j=0; j<N; j++)
    {
          population[i][j] = new_population[i][j];
    }
```

```
        }
}

void sort(float A[],int m, int order[])
{
        int i,j;
        float temp;
        for(i=0; i<m; i++)
            order[i] = i;
        for(i=1; i<m ;i++)
        {
            for(j=0;j<m-i;j++)
            {
                  if((A[j]-A[j+1])<-1e-6)
                  {
                        temp = A[j+1];
                        A[j+1]= A[j];
                        A[j]= temp;
                        temp = order[j+1];
                        order[j+1]= order[j];
                        order[j]= temp;
                  }
            }
        }
}
```